Arquitetura de software distribuído

Boas práticas para um mundo de microsserviços

Casa do Código | alura

FLÁVIO LISBOA

Sumário

- ISBN
- Agradecimentos
- Sobre o autor
- Prefácio
- 1. Introdução: nossa meta é não bagunçar
- 2. O projeto de sistema distribuído
- 3. O microsserviço de fila
- 4. O microsserviço produtor
- 5. O microsserviço consumidor
- 6. O microsserviço de monitoramento
- 7. O microsserviço agendado
- 8. Conclusão
- 9. Referências

ISBN

Impresso: 978-65-86110-86-9

Digital: 978-65-86110-87-6

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Agradecimentos

Agradeço a Deus pelos milagres e por me sustentar na dificuldade.

Agradeço à minha esposa por ser minha companheira.

Agradeço à minha filha por ser mais do que eu sonhava quando a imaginei.

Agradeço a toda a equipe da Casa do Código por tornar o projeto deste livro realidade.

Agradeço a todos os colegas desenvolvedores, aos clientes, fornecedores, professores e alunos que contribuíram com o conhecimento e experiência dos quais este livro representa um resumo.

Agradeço à minha mãe, onde quer que ela esteja, e aos que duvidaram do que eu era capaz de fazer.

Sobre o autor

Flávio Gomes da Silva Lisboa é bacharel em Ciência da Computação, com especialização em Tecnologia Java, pela Universidade Tecnológica Federal do Paraná, e mestre em Tecnologia e Sociedade pela mesma instituição.

Programador formado pelo Centro Estadual de Educação Tecnológica Paula Souza, já atuou em empresas privadas de TI e foi funcionário do Banco do Brasil, onde atuou como analista na diretoria internacional.

É analista de desenvolvimento do Serviço Federal de Processamento de Dados (Serpro), no qual foi coordenador do *Programa Serpro de Software Livre* e gerente de equipe de desenvolvimento.

Tem mais de nove anos de experiência em treinamento de desenvolvedores em Programação Orientada a Objetos, Padrões de Projeto e uso de frameworks.

Foi professor no curso de Desenvolvimento de Sistemas para Internet PHP, da UNICID.

Atualmente, é professor da disciplina Frameworks back-end em PHP do curso de pós-graduação em Desenvolvimento de Aplicações Web da UNICESUMAR e da disciplina Frameworks de Desenvolvimento PHP II na Faculdade ALFA Umuarama.

É pioneiro na bibliografia em língua portuguesa sobre Zend Framework e Symfony.

Por fim, ele é associado a *ABRAPHP*, *Zend PHP Certified Engineer*, *Zend Framework Certified Engineer* e *Zend Framework 2 Certified Architect*.

Prefácio

Como fazer **softwares que sobrevivam**?

O que queremos dizer com isso? Vamos reformular.

Como fazer softwares que sobrevivam a constantes alterações, que incluam adição de novas funcionalidades, correções de erro, evoluções provocadas por necessidade ou oportunidade e integrações com outros sistemas?

Ainda não entendeu o que queremos dizer com **sobrevivência do software**? Vamos reformular novamente.

Como fazer com que seja possível continuar dando manutenção a um software se a cada mudança ele fica mais complexo e, assim, mais difícil de compreender?

Essa pergunta ainda pode ser reescrita de uma forma que sensibilize ainda mais os que vivem da produção de software:

Como continuar entregando software como um produto ou serviço sem que o custo de sua manutenção ultrapasse o valor que ele entrega?

Essa é uma questão que deve preocupar empreendedores e gestores de projetos de softwares, e a resposta deve ser dada por um profissional denominado **arquiteto de software**.

Pode ser que não haja um arquiteto de software em sua empresa e você precise de um ou queira se tornar um. Este livro pode ser de grande ajuda nesse caso, pois ele é destinado para desenvolvedores e desenvolvedoras de software iniciantes e profissionais que atuam ou querem atuar no papel de arquitetos ou arquitetas de software.

Para aproveitar o conteúdo deste livro, você deve ter pelo menos conhecimento e experiência como iniciante em desenvolvimento de

software, independente de linguagem de programação, ou seja, você deve estar trilhando a carreira de programador. Você não deixará de ser programador ao se tornar arquiteto. Pelo contrário, se tornará um programador melhor e poderá se tornar mentor de outras pessoas programadoras. Isso não quer dizer necessariamente que você conhecerá linguagens de programação mais do que outros programadores, mas que será capaz de coordenar e aproveitar melhor o conhecimento de seus colegas.

O principal motivo de escrever este livro foi a constatação da ausência de literatura em língua portuguesa sobre arquitetura de software adequada à cultura brasileira. Existem diversos livros sobre arquitetura de software em inglês, mas esse conteúdo pode parecer assustador para quem desenvolve software no dia a dia e não pensa em todas as preocupações que são expostas nessa literatura específica. Há algumas poucas traduções, mas que podem soar distantes da realidade do mercado brasileiro.

Há também livros sobre **arquitetura de linguagem**, que orientam a construção de projetos de acordo com as características e restrições de uma linguagem de programação, o que pode dirigir o pensamento de forma equivocada, como uma caixa de ferramentas que tem apenas um martelo.

O objetivo deste livro é ser um guia prático para arquitetura de software baseado em exemplos. Conforme apresentarmos os problemas e implementarmos as soluções, abordaremos os conceitos. Ele está organizado da seguinte maneira:

No primeiro capítulo, trazemos conceitos de arquitetura a partir de um problema real de projeto de software.

O segundo capítulo apresenta o projeto de um sistema de software baseado em microsserviços para auditoria. Nesse capítulo, apresentamos o problema que pretendemos resolver e qual a arquitetura que será usada para implementar a solução.

O terceiro capítulo apresenta a implementação de uma fila em Java e discute a questão do uso de padrões, reúso e decisão entre implementar e usar algo existente.

O quarto capítulo apresenta a implementação de um microsserviço em Go, que produz dados para serem gravados na fila.

O quinto capítulo apresenta a implementação de um microsserviço em Python, que lê dados da fila para enviar a um sistema externo.

O sexto capítulo apresenta a implementação de um microsserviço em PHP, que monitora os microsserviços de produção e o consumo de dados da fila.

O sétimo capítulo apresenta a implementação de um microsserviço em JavaScript, que é executado em intervalos previamente configurados e funciona como ponto de controle para a monitoração.

Observe que trabalharemos com um software distribuído, formado por programas escritos em várias linguagens. Essa é a realidade do paradigma de microsserviços, que é a tendência contemporânea de construção de software. O arquiteto de software contemporâneo não deve ser arquiteto de uma linguagem só, mas um desenvolvedor de software poliglota. Existem vantagens em usar mais de uma linguagem de programação na construção de um sistema distribuído, e comentaremos essas vantagens ao longo do livro.

Todos os exemplos de código-fonte deste livro estarão disponíveis no GitHub em https://github.com/fgsl. O nome específico de cada trecho ou projeto será mencionado no respectivo capítulo.

Boa sorte e vamos em frente!

Capítulo 1
Introdução: nossa meta é não bagunçar

"*O esforço para tornar nosso programa mais eficiente em termos de tempo, no entanto, nunca deve ser uma desculpa para bagunçar tudo.*" – Edsger W. Dijkstra.

Um dos editores da revista *Fortune*, Gene Bylinsky, uma vez afirmou que as operações de uma fábrica pareciam organizadas até o momento em que alguém precisasse descrevê-las para um computador. Nesse momento, as pessoas descobriam que as operações estavam desorganizadas. A fábrica funcionava, mas somente ao refletir sobre o que faziam e como faziam é que elas descobriam que as operações tinham problemas.

É como uma ponte que está para cair, mas que ainda suporta alguns carros. Para os carros que passam, a ponte está funcionando, mas não se pode esperar a ponte cair para intervir nela.

Pode haver uma visão equivocada de que a atividade de programação de computadores é naturalmente caótica. Esse tipo de visão é compartilhado por anedotas como a que se segue:

A PROFISSÃO MAIS ANTIGA DO MUNDO

Um engenheiro, um médico e um programador estavam em um bar quando surgiu uma dúvida sobre qual seria a mais velha profissão do mundo. O médico começou:

— A profissão mais velha é a medicina. Afinal, se vocês pegarem a Bíblia, verão que Deus fez a mulher de uma costela do Adão praticando uma cirurgia.

O engenheiro aproveitou o gancho e disse:

— Se vocês pegarem a Bíblia verão que, antes de ter feito a mulher, Deus fez o mundo, o que é a mais perfeita manifestação de engenharia de que se tem notícia.

O programador não aguentou e retrucou:

— Acho que vocês estão enganados. É a programação. Justamente na Bíblia, que vocês citaram, a primeira frase do primeiro livro, o Gênesis, diz assim: "No começo, era o caos...".

Essa história pode até ser engraçada, mas não deve de modo algum ser tomada como o ideal de realidade para a atividade de desenvolvimento de sistemas de software. Você precisa entender que **o caos não é o estado normal da programação de computadores**. E a pessoa ou pessoa candidata à arquitetura de software deve estar plenamente consciente disso.

Desenvolver software é mais do que copiar código-fonte do Stack Overflow. Se a pessoa que programa for apenas uma copiadora de código, ela está fadada a desaparecer realmente, como afirmam as profecias sobre o fim da atividade de programação.

Tristemente, já observei diversas vezes, em palestras e treinamentos, que algumas pessoas têm como expectativa apenas copiar códigos. Elas não querem pensar, não têm paciência e não

compreendem metáforas. E isso, além de parecer a personalidade de Drax, o Destruidor, é um problema sério para quem tem a intenção de desenvolver softwares.

Pense no seguinte. O processador de um computador faz basicamente três tarefas: calcular, comparar e copiar. A diferença entre um processador e um ser humano é que o primeiro consegue fazer essas três tarefas mais rapidamente e sem se perder.

Portanto, a pessoa programadora deve fazer mais do que o processador, para ser tão ou mais útil que ele. Na verdade, o papel da pessoa que programa, em conjunto com a arquiteta, não é produzir código-fonte, mas sim **controlar a complexidade**. O código-fonte é consequência.

Vamos então entender o que é a arquitetura de software, começando com o próprio conceito de arquitetura.

1.1 O que é arquitetura?

Poderíamos começar esta seção com uma frase do tipo "arquitetura é...", mas faremos de forma diferente. Vamos primeiro mostrar **o que não é arquitetura** ou, para não sermos tão duros, **quando se nota a ausência de arquitetura**.

Vamos contar uma história relacionada à área da tecnologia da informação. Duas pessoas montaram uma startup para um negócio em que nenhuma das duas era especialista. O negócio da empresa era um site de busca, que armazenava seus dados indexados em um banco de dados relacional. Esse banco cresceu rapidamente e o desempenho do sistema começou a cair. Vamos chamar essa empresa de Promissora (é um pseudônimo, mas essa história é real).

Um dos sócios da empresa, o que tinha formação na área de exatas, teve a ideia de dividir o banco de dados único em vários bancos menores para distribuir a carga e assim melhorar o desempenho. Essa técnica, chamada *sharding*, é amplamente utilizada por empresas que operam com milhões de usuários pelo mundo, como o Instagram, por exemplo. Em 2012, o Instagram contabilizava a inserção de 25 fotos por segundo em sua base. Isso dá mais de 2 milhões de fotos em um único dia. Imagine uma tabela que cresce 2 milhões de registros por dia!

A ideia era boa, mas para implementá-la era necessário criar um identificador de *shard* (o banco menor, que é um pedaço, um fragmento, do banco com todos os dados). Como os registros ficariam em bancos separados, quando a aplicação precisasse fazer uma consulta, ela teria que saber primeiro em que banco procurar.

Em um banco de dados distribuído com a técnica de *sharding*, há uma replicação de tabelas com a mesma estrutura. Por exemplo, vamos pensar que existe uma tabela para armazenar pedidos de compra chamada `pedidos\_de\_compra`. Todos os shards (os bancos reais) terão uma tabela com esse nome. Para saber em qual banco a consulta deve ser efetuada, podemos dar um prefixo ao nome do banco que identifique qual é o subconjunto de dados que está armazenado ali.

Por exemplo, para você entender melhor, vamos pensar que a divisão do banco em vários *shards* será feita por ano e mês. Por quê? Porque você observa que o desempenho satisfatório do banco de dados cai após um mês de inserções de registros. Então você divide o banco único em vários bancos menores, um para cada mês do ano. Quando você precisar encontrar um pedido, usará a data para selecionar o banco.

Independente da linguagem de programação utilizada, para se conectar a um banco de dados é necessário usar o nome do banco. E o nome pode ser uma variável. Assim, o código que faz a conexão

com o banco é o mesmo, independente de qual banco será utilizado, pois esse dado virá da requisição do usuário.

Nesse exemplo, a cada mês um novo banco tem de ser criado, mas a aplicação não precisa ser alterada. Ela pode consultar qualquer banco e sabe qual banco consultar, pois esse dado será fornecido pelo usuário. Observe que os shards não crescem indefinidamente. Isso quer dizer que há um limite para o crescimento de registros de tabelas, mas não há um limite para a quantidade de bancos reais.

Nossa, mas depois de dez anos, você vai ter 120 bancos de dados! Se isso se tornar um problema, você pode separar esses bancos em sistemas gerenciadores de bancos de dados diferentes. Por exemplo, um gerenciador por ano. Bem, para chegar até aqui você tem de saber que há uma diferença entre o banco de dados e o sistema gerenciador de banco de dados.

A figura a seguir sintetiza o processo de *sharding* do nosso exemplo:

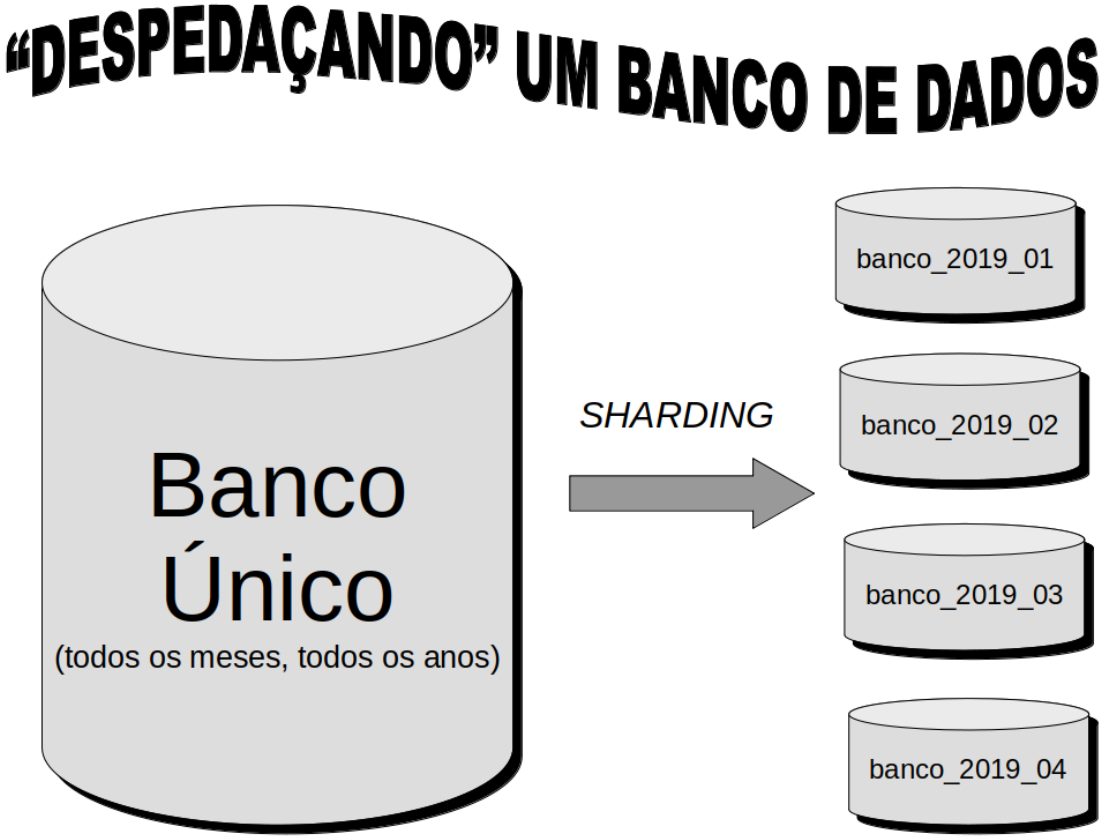

Figura 1.1: Um banco de dados distribuído com a técnica de sharding.

Maravilha! Conseguiram fazer a aplicação crescer com uma estrutura de divisão bastante simples. Você deve imaginar então que o sócio exato da Promissora (o que tinha formação em exatas) aplicou a técnica de *sharding*, resolveu o problema de desempenho do sistema da empresa e viveu feliz para sempre com seu parceiro, fazendo novos negócios. Só que não foi assim que aconteceu.

O sócio exato decidiu dividir o banco de dados de acordo com o seguinte critério: a origem dos clientes. Ele criou um banco de dados para cada estado da federação brasileira, mais o distrito federal. Um único banco se transformou em 27 bancos. Cada banco de dados tinha uma tabela de pedidos dos clientes daquele estado. Mas o sócio exato não considerou que a quantidade de pedidos de cada estado tinha um grande desvio em relação à média.

A consequência foi que o banco de dados que armazenava os pedidos dos clientes do estado de São Paulo, por exemplo, tinha quase dez vezes mais registros que outros estados, como Acre e Roraima. A figura a seguir ilustra como os *shards* ficaram desproporcionais. O problema de baixo desempenho não ocorria mais no acesso a todos os registros, mas continuava ocorrendo na maior parte deles.

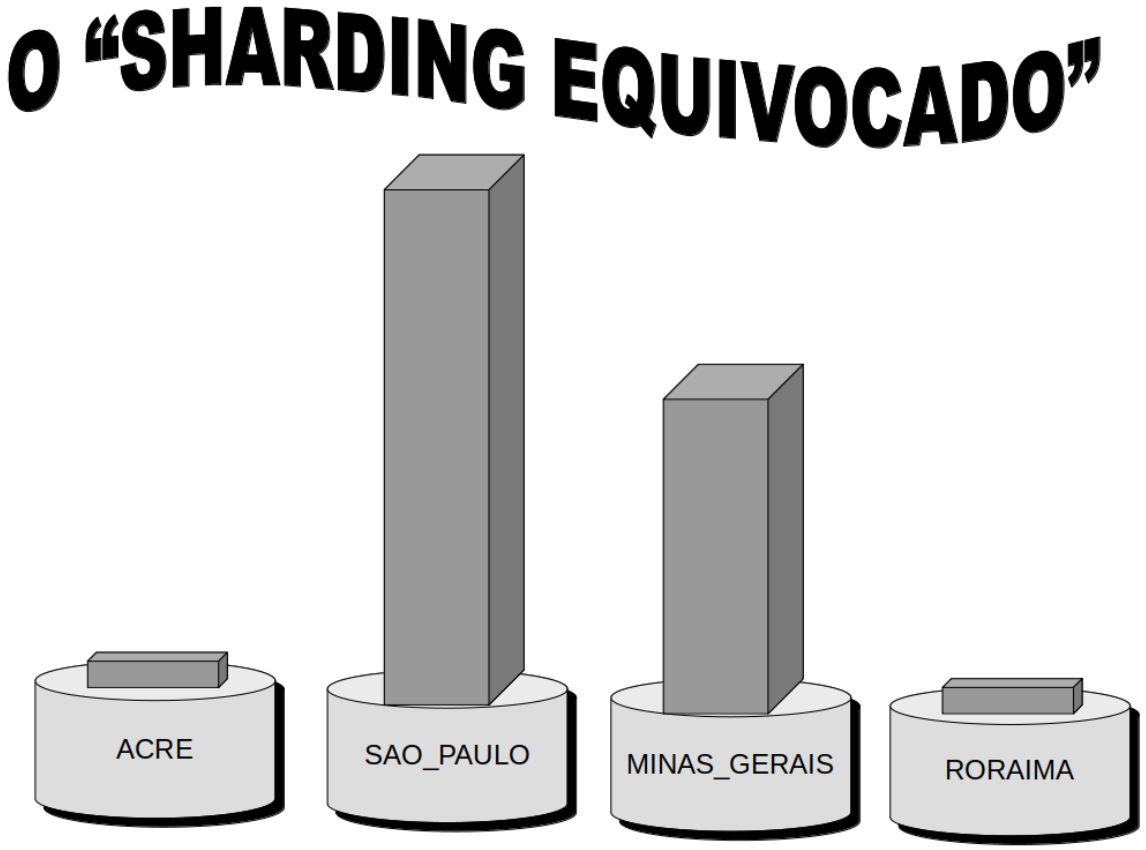

Figura 1.2: Um exemplo da consequência de um sharding inadequado.

O que essa história mostra? Algo que pode ser resumido nas palavras de Steve McConnell:

"*Sem uma boa arquitetura de software, você pode ter o problema certo, mas a solução errada. Dificilmente a construção [do software] será bem-sucedida.*"

Certo, mas a questão ainda persiste: o que é arquitetura?

Se eu fosse usar uma frase do McConnell para definir arquitetura, eu a definiria assim:

"*Arquitetura significa que você está certo de que sabe para que alvo está apontando antes de atirar.*"

No caso da empresa Promissora, o sócio exato não sabia para que alvo estava apontando.

Agora, após falar sobre o que arquitetura não é, podemos nos aprofundar sobre o que a arquitetura é.

O arquiteto romano Marcos Vitrúvio Polião escreveu em sua obra *De Architectura* o seguinte: "*Arquitetura é uma ciência que surge de muitas outras ciências, e é adornada com muitos e variados aprendizados*". Essa frase de Vitrúvio indica que arquitetura exige uma mente aberta, pois o conhecimento que ela trata é multidisciplinar.

O professor Sílvio Colin, em seu livro *Uma Introdução à Arquitetura*, nos conta que a palavra "arquiteto" tem origem no grego *tecton*, que "*designava um artífice ligado à construção de objetos por junção de peças, como um carpinteiro, e não por modelagem ou entalhe*". Então um arquiteto é um carpinteiro? Na verdade, mais do que isso. O prefixo grego *arqui* indica superioridade. O arqui-inimigo de um herói, por exemplo, é o seu maior inimigo.

Assim, podemos dizer que a palavra "arquiteto" significa, em sua origem, "*um carpinteiro superior*". Mas o sentido fundamental da palavra "arquiteto" é **alguém que constrói objetos juntando peças**.

Entre os significados da palavra "arquitetura" que podemos encontrar no dicionário Michaelis, destaco dois:

"*Arte e ciência de projetar e supervisionar a construção de edifícios ou outras estruturas [...]'.*

"*Estrutura, disposição e organização de um conjunto geralmente harmônico*".

Provavelmente, para o grande público, a palavra "arquitetura" remete diretamente à construção de edifícios, mas o dicionário indica que a arquitetura também cuida da construção de outras coisas ou de outros objetos. A preocupação da arquitetura é criar estruturas, dispor objetos, organizar e finalmente criar harmonia, ordem (e não caos).

Simon Brown, em seu livro *Software Architecture for Developers*, afirma que "*como um substantivo então, arquitetura pode ser resumida como sendo sobre estrutura*" (em tradução livre).

Eu definiria arquitetura como **"a arte de estabelecer a ordem"**. E o estabelecimento da ordem é uma necessidade em várias áreas do conhecimento.

Vamos pensar agora no tema deste livro, software. Para que fazemos softwares? Para satisfazer metas de negócios. Sistemas de informação são virtualizações de processos de negócio que já existem. Nós transportamos esses processos de negócio para computadores para que eles se tornem mais eficientes, porque eficazes eles já são.

O transporte do processo de negócio do mundo real para o virtual pode falhar se não for organizado. É preciso compreender a estrutura da empresa e relacioná-la com uma outra estrutura que operará em um computador.

Uma boa arquitetura é a ponte para essa travessia, uma ponte de pedra, larga e segura. Uma arquitetura ruim também pode ser uma ponte, mas de corda, estreita e com chance de arrebentar a qualquer momento.

Na construção de sistemas de informação, existem várias arquiteturas a serem trabalhadas. Há a arquitetura da rede, a arquitetura do armazenamento, a arquitetura de aplicativo, ou arquitetura de software, e a arquitetura de usuários. Você pode trocar tranquilamente a palavra "arquitetura" nesses termos por "necessidade de organizar ou estruturar".

A reunião dessas arquiteturas compõe a arquitetura do sistema de informação, que é a visão geral sobre a estrutura das relações entre software, hardware e humanos.

O desenho de uma arquitetura de sistemas geralmente é o que se apresenta para os clientes, pois ela mostra o que interessa diretamente para eles. As outras arquiteturas estão mais ligadas ao pessoal de suporte técnico, com exceção da arquitetura de usuários, que se refere especificamente à interação dos humanos com as máquinas.

Neste livro, o que nos interessa é a arquitetura de aplicativo, ou arquitetura de software. Essa arquitetura se refere ao conjunto de estruturas que nos permite compreender os elementos que compõem um software, como esses elementos se relacionam entre si e quais são as propriedades desses elementos. Um software é composto por diversas estruturas e pode se tornar extremamente complexo.

Frederick P. Brooks (2009) afirma que há quatro níveis de complexidade de software. O primeiro é o software constituído por um único programa, que é dominado completamente por um único programador ou programadora. Nesse caso, toda a estrutura do software é conhecida por uma pessoa. Mas esse software constituído por apenas um programa geralmente é usado apenas pela pessoa que o programou e ninguém sabe que ele existe além do seu criador ou criadora. Quando as demandas crescem e a pessoa que programou esse software não consegue mais dar conta de atender a todas as alterações necessárias, esse programa se

torna um programa-produto e passa a ser utilizado por outras pessoas.

Sendo assim, essa segunda categoria de software já exige que a pessoa programadora interaja com outras pessoas para criar o programa, ou programas, para que ele seja operável por qualquer um. Assim como uma andorinha só não faz verão, um programa só não resolve todos os problemas de uma organização.

Dessa forma, é necessário criar vários programas para produzir um software que resolva um problema mais complexo, que interesse e envolva várias pessoas de uma empresa. Este software, que agora pertence à terceira categoria, já é um sistema de programas, uma coordenação de várias estruturas que seriam, isoladas, os programas operados diretamente por quem programa.

A dificuldade de criar e manter esse sistema de programas já exige uma equipe. Esse sistema de programas eventualmente pode crescer e se integrar com outros sistemas de programas, para compartilhar dados e conectar diferentes unidades da empresa ou diferentes empresas. Aí ele se torna a quarta categoria, o produto da programação de sistemas, o mais alto nível de complexidade que um software pode atingir.

Um problema que ocorre constantemente em empresas é a dificuldade de reconhecer o momento em que um software exige uma equipe. Os problemas aparecem, demandam soluções e as pessoas programadoras ou técnicas de suporte que também sabem programar criam programas para resolvê-los.

Enquanto eles são programas executados apenas por pessoas técnicas, é como se não existissem. Para quem abre um chamado para uma pessoa da área técnica, não importa como ela resolveu o problema, se usou um programa ou não. Essa pessoa quer um resultado, uma solução. A manutenção do software constituído por um programa é responsabilidade exclusiva de quem o criou. Se essa pessoa sair da empresa e não tiver documentado esse

programa, ninguém mais saberá usá-lo. Mas como ninguém sabia que o programa existia além de quem programou, ninguém sentirá preocupação.

Arquitetura de software envolve não somente ter um software funcionando, mas um software cuja estrutura esteja documentada e seja conhecida por mais de uma pessoa. E não basta ser conhecida, ela tem de ser compreendida.

É por isso que ela tem de ser organizada de tal forma que se pareça com uma cidade na qual você consegue transitar pelas ruas com tranquilidade, encontrando o caminho para os locais onde você deseja chegar.

1.2 O que vem a seguir

Neste livro, discutiremos sobre arquitetura de software na prática. No próximo capítulo, apresentaremos um projeto de software que será implantado em um ambiente distribuído, segundo o paradigma dos microsserviços. Se você não sabe o que é um microsserviço, não se preocupe, pois explicaremos isso. Vamos em frente!

Capítulo 2
O projeto de sistema distribuído

"*A simplicidade tende ao desenvolvimento; a complexidade, à desintegração.*" – Peter Drucker.

Atualmente, há uma grande disponibilidade de manuais de linguagens de programação. A possibilidade de distribuir documentação em formato digital tornou muito mais fácil encontrar referências sobre as funcionalidades de softwares e hardwares. Não há necessidade de listar todos os itens de configuração ou atributos de um software em um livro, porque a documentação aberta em um website já permite uma busca rápida.

É por isso que aqui não faremos um manual de propriedades de um sistema distribuído, mas daremos orientações sobre como resolver problemas recorrentes em sistemas distribuídos. Isso será feito a partir de um exemplo, mas as observações que faremos podem ser reproduzidas para diferentes sistemas, porque destacaremos padrões de solução.

Vamos apresentar neste capítulo um problema que resolveremos com a construção de um sistema distribuído. É importante, em uma visão de arquitetura de software, que você concorde que:

- Um problema pode ter mais de uma solução;
- Há soluções melhores do que outras;
- Você pode não entender o problema inicialmente.

2.1 Aprendendo com os erros

O problema que motivará nosso projeto de sistema distribuído baseado em microsserviços pode ser resolvido de outras maneiras.

Espera-se, na verdade, que alguém sempre encontre uma maneira melhor de fazer algo. Isso é evolução. A busca por algo melhor deve fazer parte do trabalho da arquiteta e do arquiteto. Entretanto deve-se reconhecer que só é possível melhorar algo que existe. A ideia sobre uma nova forma de resolver um problema só é possível porque alguém o resolveu da forma atual. O legado de soluções permite progredir. Não há progresso sem qualquer referência inicial.

Eventualmente, podemos descobrir que um problema não foi resolvido. Quer dizer, alguém criou uma solução acreditando que estava resolvendo um problema, mas essa mesma pessoa, ou outra, mais tarde descobre que o problema não havia sido analisado minuciosamente e, por isso, alguns aspectos críticos foram ignorados.

Há uma categoria de problema destacada por McConnell (2005) que é o *problema perverso*. Esse problema é aquele que você só compreende depois de tentar resolvê-lo uma primeira vez. A tragédia do problema perverso é que você geralmente acredita que o entendeu e assume que a solução que encontrou é a definitiva, dispensando simulações e testes.

Na literatura sobre arquitetura de softwares, há dois exemplos frequentemente citados de problemas perversos: o navio de guerra sueco Vasa e a ponte Tacoma Narrows.

O primeiro era o maior navio de guerra que já havia sido construído até então. O desconhecimento sobre o efeito do peso de uma fileira dupla de canhões em um navio daquele porte fez com que o Vasa afundasse logo após ser lançado ao mar.

A ponte Tacoma Narrows era uma ponte suspensa sobre um rio. O desconhecimento sobre os efeitos do vento na ressonância da ponte fez com que ela fosse despedaçada durante um vendaval.

Nos dois casos, foi necessário tentar resolver os problemas para descobrir que eles não haviam sido compreendidos totalmente. E o

preço disso foi bastante alto, principalmente no primeiro caso, em que parte da tripulação morreu.

O sistema que vamos criar já passou pela fase da primeira tentativa de solução e, como vimos, isso não quer dizer que a solução atual é a definitiva. Antes de falar da solução, temos que apresentar o problema, porque, afinal de contas, o problema precede a solução no mundo real. Além disso, você precisa ter a chance de imaginar a sua solução antes de apresentarmos a nossa, para que não fique a impressão de que estamos ignorando suas ideias.

Dito isso, vamos a um cenário comum para o desenvolvimento de software contemporâneo.

2.2 Nosso projeto de sistema distribuído

Uma empresa, que vamos chamar de Nuvem S.A, vende serviços de infraestrutura de computação em nuvem. O negócio envolve o aluguel de servidores para a instalação de aplicações (de software), com uso de clusters de computadores para fornecer servidores virtuais. De forma mais específica, essa empresa usa um gerenciador de clusters chamado Kubernetes.

Kubernetes permite gerenciar clusters e provê contêineres no lugar de máquinas virtuais completas. Isso quer dizer que é possível oferecer um espaço para uma aplicação com tudo aquilo de que ela precisa e nada mais, evitando que recursos desnecessários sejam alocados.

Contêineres são pedaços de sistemas operacionais, que usam apenas uma parte dos recursos computacionais disponíveis. Além de racionalizar o uso de recursos, os contêineres ajudam na segurança, ao evitar que uma aplicação tenha acesso a objetos que não fazem parte do seu negócio.

Uma das grandes vantagens de Kubernetes é que ele permite reiniciar conjuntos de contêineres, que ele chama de pods, sem intervenção humana. Basta que você configure verificações de estado e ele reiniciará os pods. Isso evita que a aplicação fique indisponível, se um problema exigir apenas uma reinicialização. Nenhuma pessoa técnica precisa ser avisada e os usuários e usuárias podem nem perceber que algo foi reinicializado, de tão rápido que a operação aconteceu. Afinal, não é um computador físico que está sendo reiniciado, mas um conjunto de contêineres em uma máquina virtual.

Vamos tentar resumir o funcionamento do Kubernetes. Você instala sua aplicação em um contêiner que tem todos os componentes de que ela precisa para funcionar. Esse contêiner está dentro de uma estrutura chamada pod, que é gerenciada pelo Kubernetes, de modo que ele pode reiniciar o contêiner, retornando a aplicação a um estado inicial.

Na verdade, a questão da reinicialização é feita por uma estrutura maior chamada deployment, que é um conjunto de pods. O deployment é o que pode garantir a disponibilidade, mas ele não garante isso sozinho.

O fato de o Kubernetes poder reiniciar seus pods de forma automática não significa que todos os problemas estão resolvidos. Pode ser que a aplicação tenha um problema que não seja sanado com uma reinicialização, então o Kubernetes vai ficar tentando reiniciar indefinidamente, e a aplicação continuará indisponível. É necessário monitorar as mudanças que ocorrem com os pods para saber se é necessária uma intervenção. Também é necessário manter um registro de mudanças para o caso de uma auditoria do sistema.

Nosso sistema de auditoria - Podips

Aqui entra o sistema que vamos construir, o Podips. O Podips é um sistema de auditoria de pods para Kubernetes. Ele monitora as

mudanças de IPs em pods. Quando um pod é criado, ele recebe um endereço IP, para que seja possível a comunicação em rede. Esse endereço pode ser modificado se o pod for reiniciado. Assim, as mudanças de IP podem indicar problemas em aplicações instaladas em clusters Kubernetes.

O objetivo do Podips é coletar os eventos de mudança de IP do Kubernetes e registrá-los em um sistema de gerenciamento de logs que permita uma fácil consulta a esses dados. Nosso sistema precisa ler dados de um computador, que é o servidor principal do Kubernetes (o master), e gravá-los em outro, onde está instalado o sistema de logs.

Você pode pensar que esse sistema de auditoria é muito simples. É da categoria 1 de complexidade de software segundo Brooks, segundo a qual o problema pode ser resolvido com um programa que é criado e conhecido por apenas uma pessoa programadora.

Realmente, é possível fazer a leitura dos eventos e a gravação no servidor de logs em um único programa, e uma única pessoa pode fazer isso, mas se o programa de auditoria falhar perdemos dados. Então, como garantir que nosso sistema funcione o tempo todo?

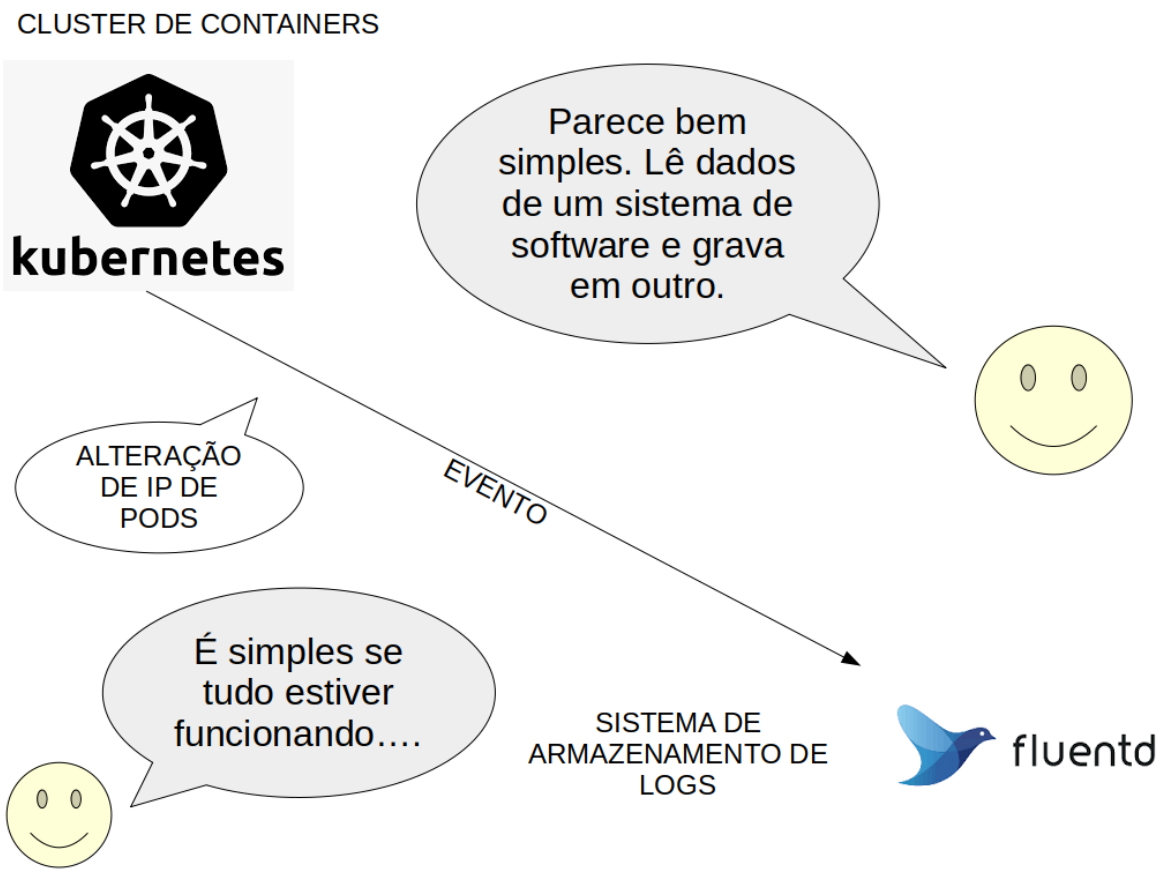
CLUSTER DE CONTAINERS
kubernetes
Parece bem simples. Lê dados de um sistema de software e grava em outro.
ALTERAÇÃO DE IP DE PODS
EVENTO
É simples se tudo estiver funcionando....
SISTEMA DE ARMAZENAMENTO DE LOGS
fluentd

Figura 2.1: O problema a ser resolvido e sua aparente simplicidade.

Para evitar que os dados de mudança de IP sejam perdidos, não vamos fazer a leitura e a gravação em sequência imediata. Um programa (**podips-reader**) fará apenas a leitura dos eventos gerados pelo Kubernetes e outro (**podips-writer**) fará apenas a gravação no servidor de logs. Entre esses dois programas, haverá uma fila.

Uma fila é uma estrutura de dados básica da computação, que permite que guardemos dados para processar posteriormente. Nós a usaremos para garantir uma parte da disponibilidade da nossa aplicação.

Resgataremos também um conceito que é usado desde os primórdios da programação para computadores eletrônicos, que é o

programa monitor, aquele que acompanha a execução do programa principal. A monitoração é essencial para garantir a disponibilidade de um sistema de informação. O programa monitor do nosso sistema será o **podips-monitor**.

Finalmente, temos o elemento de controle, um programa que serve para garantir as condições mínimas de funcionamento de um outro programa ou sistema de programas. Esse programa, o **podips-cronjob** será de execução agendada, ou seja, ele será executado recorrentemente em intervalos determinados.

Fomos levados assim, pelos requisitos não funcionais do sistema, a distribuir suas responsabilidades entre vários programas. Isso quer dizer que nosso sistema de auditoria é um sistema distribuído. Pode ser conveniente agora entender melhor o que é um sistema distribuído para compreendermos nossa decisão.

2.3 Sistema distribuído

Segundo Tanenbaum e Steen (2007, p. 1) um "sistema distribuído é um conjunto de computadores independentes que se apresenta a seus usuários como um sistema único e coerente". Para se apresentar como um sistema único, esses computadores precisam estar conectados por meio de uma rede de comunicação. Essa rede pode ser composta por cabos, torres de transmissão e recepção e até satélites.

O aplicativo de celular que mostra para você o mapa da cidade e indica o caminho que você deve seguir para chegar a um destino é um sistema distribuído. O celular que tem esse aplicativo é um computador. Ele tem um microprocessador e um sistema operacional instalado. O aplicativo de mapas envia e recebe dados usando a rede móvel ou Wi-Fi. Por meio dessa rede, ele se

comunica com um ou mais computadores que processam os dados enviados.

Há vários programas em computadores diferentes que cooperam entre si para entregar a informação que você precisa. Mas para você, na prática, há um único sistema, que é a interface do aplicativo com o qual você interage. O que ocorre entre os computadores que compõem o sistema de mapas é ignorado na maior parte do tempo pelo usuário.

Por que o aplicativo de mapas é um sistema distribuído? Ele não poderia funcionar somente no celular? Poderia, se ele processasse apenas dados armazenados localmente ou que fossem captados diretamente pelo dispositivo. O aplicativo de mapas combina dados de várias fontes, incluindo dados de governos locais, o que exige que ele se conecte a outros computadores que nem fazem parte da infraestrutura computacional da empresa que mantém o aplicativo.

Além disso, há uma série de dados que precisam ser atualizados constantemente, como pontos comerciais e seus horários de funcionamento. Isso exige consulta a uma base de dados centralizada, que consiga garantir que todos os dispositivos móveis terão os mesmos dados atualizados. O mapa deve ser o mesmo para qualquer um que esteja usando o aplicativo, em qualquer lugar. Para garantir a uniformidade de informações, é preciso processá-las de forma centralizada e distribuí-las.

Em resumo, distribuímos um sistema quando ele precisa ser utilizado por vários usuários, em localidades diferentes, por diferentes dispositivos e todos precisam compartilhar os mesmos dados. A distribuição de um sistema também ocorre quando isso é necessário para fazê-lo funcionar ou quando isso torna seu funcionamento melhor.

Agora que explicamos de forma sumariada a escolha por um sistema distribuído, podemos falar com mais profundidade do

problema apresentado anteriormente e mergulhar na implementação da solução.

Você pode ter eventualmente ficado decepcionado pelo projeto não ser um cadastro, mas um cadastro com sistema distribuído pode ser construído com apenas dois computadores, como ocorre com um simples sistema web no qual parte roda no navegador e parte roda em um servidor web. Essa arquitetura é muito simples e foi pensada para ser assim. Queremos apresentar algo mais complexo, que possa ser aproveitado em desafios maiores de sistemas que tendem a crescer e se integrar a outros.

Aqui nós lidaremos com pelo menos cinco computadores (virtuais), o que deixará nossa aventura mais interessante.

A partir do próximo capítulo, implementaremos nossa solução. Eventuais dúvidas sobre o problema serão sanadas conforme apresentarmos e explicarmos o código e, inclusive, ao discutirmos as alternativas de implementação.

Capítulo 3
O microsserviço de fila

Pontual é alguém que resolveu esperar muito. – Millôr Fernandes.

As revoluções industriais, que ocorreram do final do século XIX ao início do século XXI, forneceram para a humanidade um conjunto de tecnologias que permitiu reduzir o tempo de deslocamento físico entre lugares, ao mesmo tempo em que criou e expandiu o acesso à comunicação instantânea. Um evento ocorrido em Vladivostok, no extremo leste da Rússia, pode imediatamente ser conhecido por uma pessoa na Patagônia argentina. Temos acesso hoje a uma quantidade quase ilimitada de dados, produzidos continuamente, a partir de aparelhos que cabem em nossas mãos. E também nos tornamos disponíveis a qualquer momento.

Nesse cenário, somos impelidos a querer respostas imediatas para tudo, o que nem sempre é possível já que algumas respostas não estão prontas, pois elas precisam ser construídas ou descobertas. É necessário tempo para obtê-las. No caso de respostas obtidas com o auxílio de computador, temos o tempo de processamento. Quando a resposta de um computador depende da cópia de um dado de um lugar para outro, o tempo de resposta dependerá essencialmente da velocidade de transmissão da rede.

É como transportar um objeto de um lugar para outro. Mas, se a resposta depende de um processamento que envolve análises e cálculos, não basta ter cabos de fibra óptica para ter uma resposta rápida. E se esse processamento envolve a integração entre processamento de vários computadores, o tempo de resposta pode ser ainda maior.

Algumas tarefas de computação exigem segundos, outras, minutos, outras exigem horas e há ainda tarefas que levam dias para serem executadas.

A espera não depende somente do tempo necessário para o computador executar as instruções de um programa. Ela depende também da concorrência de usuários por um recurso computacional. Um processador pode ser compartilhado por vários usuários, por meio de uma estratégia de escalonamento gerenciada por um sistema operacional. Ainda assim, cada processador só pode executar uma instrução de cada vez, por mais rápido que faça isso. O tempo do processador é um recurso computacional, mas não é o único. Os periféricos conectados ao computador também são recursos computacionais.

Estamos em um momento em que "transformação digital" virou a expressão da moda. Não há nada de novo em seu significado, que é o de promover a automação de processos manuais e digitalização de documentos físicos. A transformação não começou agora, mas tornou-se mais intensa. O uso de papel torna-se cada vez mais desnecessário com a disponibilidade de dados em formato digital. Com um aparelho celular você pode ler um documento produzido em qualquer lugar do mundo, sem a necessidade de transporte físico. A impressora está se tornando um item de uso mais pontual, pois não é necessário imprimir um documento e assinar para entregá-lo, é possível assiná-lo digitalmente. Isso evita gastos com papel e contribui para reduzir o desmatamento.

Mas a impressora ainda continua sendo utilizada e geralmente, em empresas, é um recurso compartilhado. Não há uma impressora para cada funcionário, a não ser que seja um gerente ou um empregado com alguma necessidade específica. Assim, uma impressora é conectada a uma rede de computadores e todos os funcionários podem enviar trabalhos para impressão por meio de seus dispositivos conectados a essa rede. Quando os funcionários enviam documentos para impressão em momentos diferentes, não há problema nenhum. Mas e quando dois funcionários enviam seus documentos ao mesmo tempo? O que a impressora faz? Ela não pode imprimir os dois documentos simultaneamente, ela processa um trabalho de cada vez. Os documentos, na verdade, não são

enviados diretamente para a impressora, eles ficam esperando em uma **fila**.

A fila em computação é uma estrutura de dados. É uma forma de organizar uma parte da memória de um computador para um objetivo específico. O objetivo específico da fila é armazenar dados que não serão processados imediatamente. Como na fila de um banco, em que há um caixa e dez pessoas para serem atendidas, não há como um computador atender a todas as demandas de forma imediata.

A fila computacional permite armazenar os dados de uma demanda e processá-los quando for possível. Para isso, a memória é dividida em unidades, que são os nós da fila, e uma disciplina de acesso é estabelecida. Os primeiros dados a entrarem na fila são os primeiros a saírem dela, assim como a primeira pessoa a entrar na fila do banco é a primeira a ser atendida pelo caixa.

Ah, mas e os clientes preferenciais? Eles não passam na frente da fila? Sim. Por isso, a fila também prevê a priorização de trabalhos na ordenação dos dados que são armazenados. Quando um dado é colocado em uma fila com priorização, ela é reordenada de acordo com a prioridade do dado que entrou. Dados com a mesma prioridade são enfileirados pela ordem de chegada, de modo que, se todos tiverem alta prioridade, será como se não tivessem prioridade nenhuma. É como se apenas idosos estivessem na fila do banco. Se apenas a idade for critério para priorização, ninguém será priorizado neste caso.

3.1 A estrutura de filas no nosso sistema de auditoria

Acredito que a ideia geral da estrutura computacional de fila tenha ficado clara. Mas o que ela tem a ver com o nosso projeto de

sistema de auditoria de pods?

Bem, os pods recebem endereços IP quando são criados e podem ter os endereços alterados se forem reiniciados. A criação e a alteração dos pods não é um evento determinístico. O que isso quer dizer? Não temos um planejamento absoluto de quando isso vai acontecer. Como assim, as pessoas desenvolvedoras não têm um planejamento de quando publicarão as versões de seus sistemas no cluster Kubernetes? Provavelmente, sim. Elas podem fornecer a data planejada para lançamento de versões, mas, se as únicas alterações ocorressem nessas datas, talvez não precisássemos de uma fila, o que não ocorre na realidade. A realidade do mundo do software em produção é bastante dinâmica.

Além das datas planejadas para lançamento de novas versões, podem ocorrer lançamentos não planejados para corrigir bugs. Esses podem até fazer parte de um planejamento de correção, mas que não era parte do planejamento original, ou seja, nosso sistema de auditoria não os levaria em conta.

Nós também comentamos que uma das vantagens de se usar um cluster Kubernetes é sua capacidade de reiniciar os pods sozinho de acordo com condições programadas, para tentar garantir a disponibilidade contínua do sistema. Você informa uma URL que indica se a aplicação está funcionando e, se ele não responder, o Kubernetes reinicia o pod, restaurando as condições iniciais de funcionamento. Pode ser que isso não resolva o problema para todos os casos, mas evita que a equipe de suporte seja acionada só para reiniciar uma máquina virtual (ou neste caso um contêiner, que é algo menor ainda). O Kubernetes automatiza uma parte do suporte técnico e, quando ocorrem essas reinicializações, há mudança nos IPs dos pods. E não sabemos quando elas ocorrerão, porque a condição de indisponibilidade é probabilística.

Isso quer dizer que em um momento você pode ter uma série de eventos de alteração ocorrendo em grande quantidade e em outro momento, nenhum evento. Se seu cluster hospeda dezenas de

aplicações e todas elas têm condições de indisponibilidade com probabilidade maior que zero, podem ocorrer dezenas de eventos de alteração de IP em poucos segundos. É como um dia de pagamento em que muita gente vai ao banco para sacar dinheiro. Não importa se é no caixa dentro da agência ou no caixa automático. Se há mais pessoas que pontos de atendimento, algumas terão de esperar pelas outras. Então, quando os eventos de alteração forem criados, eles serão guardados em uma fila, para uma leitura gradual e envio para o sistema de armazenamento de logs.

Existem vários sistemas de fila disponíveis no mercado, tanto para instalação em uma infraestrutura própria como para consumo como serviço. Alguns são pagos e alguns são gratuitos. Entre os gratuitos, alguns têm o código aberto: você tem acesso ao código-fonte e pode não apenas usar como modificar, melhorando a implementação. O sistema de fila que utilizaremos em nosso projeto é o Apache ActiveMQ, disponível em https://activemq.apache.org.

3.2 Instalando o Apache ActiveMQ

Em seu site, o Apache ActiveMQ é descrito como um "intermediário de mensagens". Ele implementa filas seguindo um protocolo denominado AMPQ (*Advanced Message Queueing Protocol*), cuja documentação está disponível em https://www.amqp.org. Um arquiteto de software deve preferir produtos de software que implementem padrões, principalmente padrões abertos. Isso abre a possibilidade de troca desse produto por outro similar em algum momento, por obsolescência ou por novos requisitos.

Não queremos ficar mudando os produtos que usamos aleatoriamente. Mas queremos ser capazes de mudar se isso se mostrar necessário. Ao adotarmos um produto que usa o AMPQ, seremos capazes de substituí-lo por outro que use o mesmo

protocolo sem que as outras partes do sistema precisem ser alteradas. A redução do impacto de alteração é uma meta a ser alcançada pela pessoa arquiteta de software nas suas decisões sobre a estrutura de um sistema.

Nós instalaremos o Apache ActiveMQ "Classic", que nos atenderá com relação aos propósitos didáticos. Baixe o arquivo de instalação compatível com seu sistema operacional a partir de https://activemq.apache.org/components/classic/download. Se você precisar de alta performance da fila em ambiente de produção, pode instalar, em outro momento, o ActiveMQ Artemis.

O arquivo que você baixará é uma aplicação Java. Por isso, o pré-requisito para o funcionamento do ActiveMQ é a instalação da máquina virtual Java (JVM). Você deve conferir a versão mínima de Java, assim como requisitos de hardware e versão de sistema operacional na página de instruções da instalação: https://activemq.apache.org/getting-started. Para a versão 5.16 do ActiveMQ, a JVM mínima é a 1.7.

Descompacte o arquivo em um diretório de sua preferência. Abra esse diretório no terminal de seu sistema operacional e entre no subdiretório `bin`. Dentro desse subdiretório, há um script de inicialização, o `activemq` (ou `activemq.bat` para Windows).

Após iniciar o ActiveMQ, você poderá administrá-lo por uma aplicação web por meio do endereço http://127.0.0.1:8161/admin. É necessário se autenticar para acessar essa aplicação. O usuário padrão é *admin* e a senha é *admin*. Se você quiser mudar a senha do administrador e criar outro usuário, basta editar o arquivo `users.properties`, que fica no subdiretório `conf` do ActiveMQ.

Se tudo der certo, você verá a página da imagem a seguir.

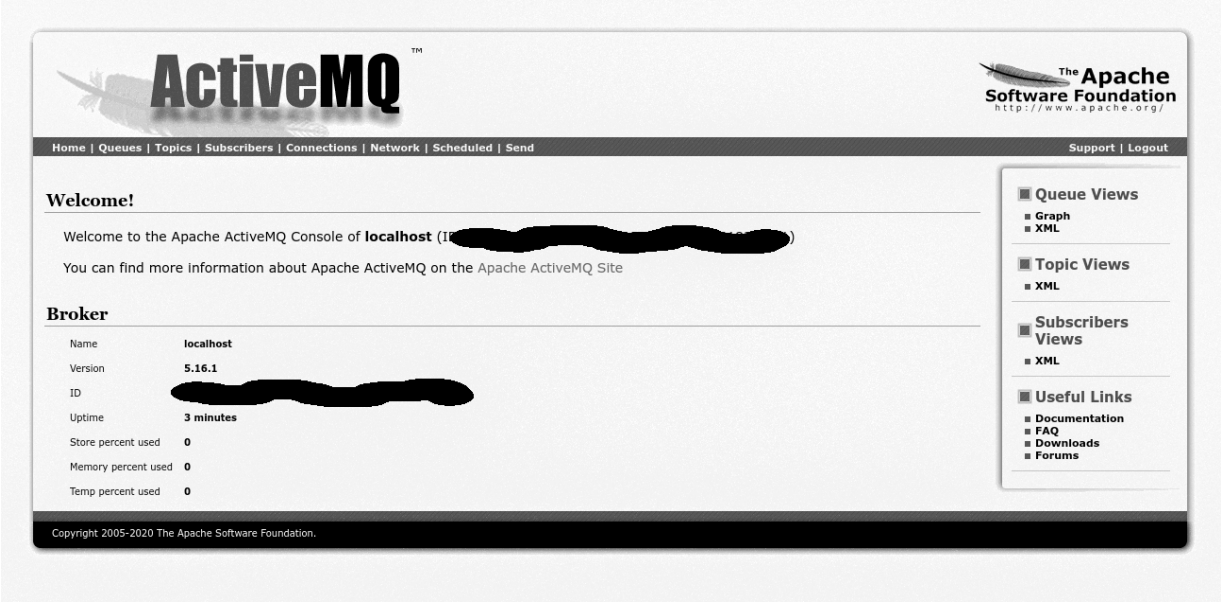

Figura 3.1: Página inicial do administrador web do ActiveMQ.

Criando a fila do sistema Podips

O ActiveMQ é um sistema gerenciador de filas. Você, como administrador, define as filas. Vamos criar uma nova fila. Clique no item `Queues`, que fica no menu horizontal. Ele é o segundo item da esquerda para a direita. Ao clicar nesse item, será exibida a página de filas com um formulário para criar novas filas, conforme a próxima imagem. Vamos criar a fila que será usada pelo nosso sistema, o Podips. Digite o nome `pods` no campo `Queue name` e clique no botão `Create`.

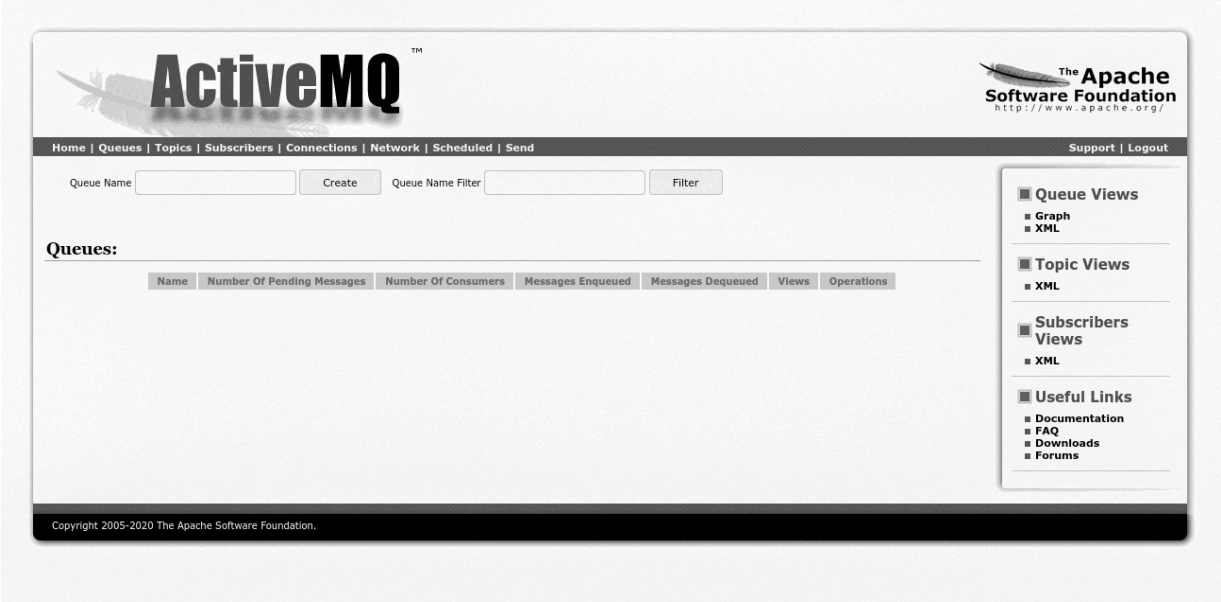

Figura 3.2: Página de filas do ActiveMQ web.

Após criar a fila, ela passa a ser exibida na lista de filas abaixo do formulário, como podemos ver na imagem seguinte. Nessa lista, podemos ver os indicadores de mensagens. Temos a quantidade de mensagens pendentes (`Number Of Pending Messages`), que são as mensagens que se encontram atualmente na fila, o número de consumidores (`Number Of Consumers`), que são os programas que leem da fila, o número de mensagens enfileiradas (`Messages Enqueued`), que é o total de mensagens que foram gravadas na fila, e finalmente o número de mensagens desenfileiradas (`Messages Dequeued`), que é o total de mensagens que foram lidas da fila. Para cada fila, há hyperlinks do lado direito, que permitem enviar mensagens para a fila (`Send to`), limpar a fila (`Purge`), apagar a fila (`Delete`) e parar a fila (`Pause`).

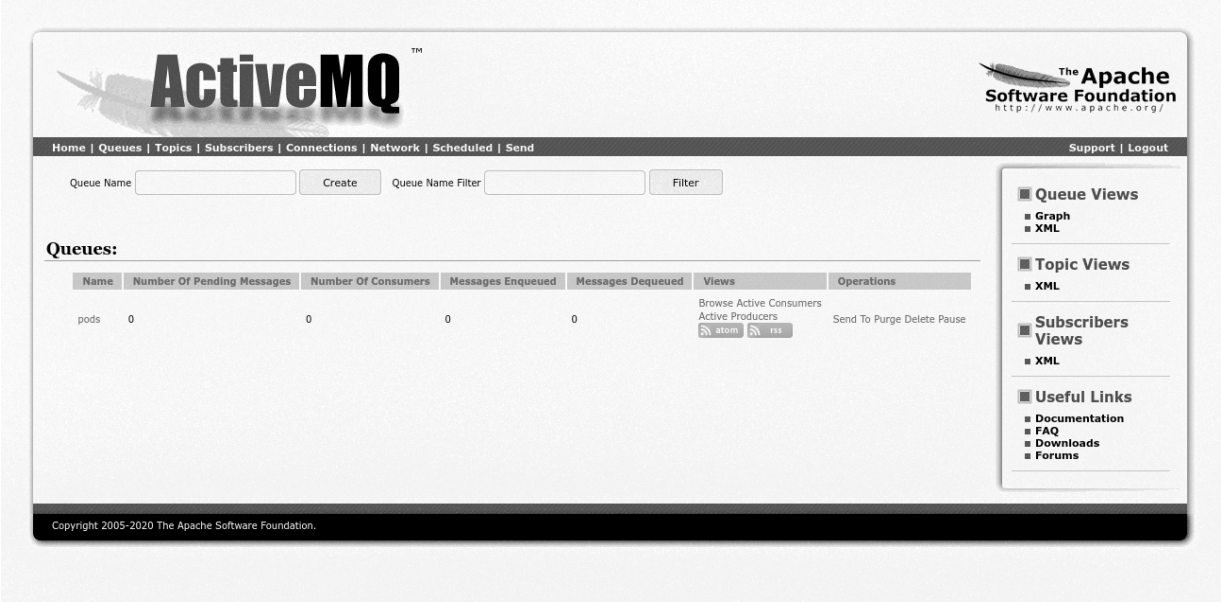

Figura 3.3: A fila pods criada e aguardando mensagens.

Agora que a fila está criada, vamos experimentar enviar uma mensagem para ela. Clique no link `Send To` e será apresentada uma página com um formulário, como mostra a próxima imagem. Na área de texto `Message body`, escreva um texto qualquer e clique no botão `Send`. Você será direcionado para a página de filas e verá que o indicador de mensagens na fila foi alterado para 1.

Figura 3.4: O formulário de envio de mensagens.

Temos um sistema de gerenciamento de filas em funcionamento baseado em um software livre e aberto, escrito na linguagem Java. Agora as mensagens geradas pelo sistema Podips já têm um local para serem armazenadas enquanto transitam entre a fonte e o destino final. Vamos fazer a seguir uma simulação do funcionamento da fila em um processo contínuo de gravação e leitura de mensagens.

3.3 Simulação de produtor e consumidor

O sistema produtor

Uma fila, para ter utilidade, precisa de dois atores: um produtor, que grava mensagens, ou as põe na fila, e um consumidor, que lê as mensagens, ou as tira da fila. Esse é um detalhe que temos de lembrar: a leitura de dados em um fila remove esses dados da fila. Não é como um banco de dados onde você lê um registro de uma tabela e ele permanece lá. É por isso que, quando lermos a mensagem da fila, precisamos garantir que ela seja persistida em seu destino final. Essa questão não será tratada aqui. Neste momento, queremos saber se a fila consegue interagir com os dois atores necessários para sua utilidade.

Vamos usar programas que simulam os sistemas reais para os quais o Podips está sendo construído. Primeiro vamos instalar um sistema que simula a produção de eventos do Kubernetes. Esse sistema é o `mock-producer`. Você pode instalá-lo usando o **Git** ou baixá-lo como um arquivo ZIP.

Se você não usa o Git, então está completamente alienado ou alienada do mundo do desenvolvimento de software do século XXI. Git não é algo opcional para desenvolvedores e desenvolvedoras atualmente. Então, se você deseja se tornar uma pessoa desenvolvedora, aproveite a oportunidade para começar a usar uma ferramenta altamente essencial, que é um sistema de controle de versões distribuído. Para instalar o Git, visite o site https://git-scm.com/downloads.

Se você optar por instalar o `mock-producer` com o Git, você executará o seguinte comando no terminal, em um diretório de sua escolha:

```
git clone https://github.com/fgsl/mock-producer.git
```

Caso contrário, basta descompactar o arquivo ZIP, disponível no botão `Code` da página https://github.com/fgsl/mock-producer, em um

diretório de sua escolha.

Em qualquer uma das opções, você terá como resultado a criação de um diretório `mock-producer`. Dentro desse diretório, há um programa escrito em linguagem Go, o `mock-producer.go`. Para usá-lo, precisamos compilá-lo, e para isso precisamos do compilador Go, que você pode instalar a partir desta página: https://golang.org/dl. Após instalar o compilador Go, acesse via terminal o diretório `mock-producer` e execute o comando a seguir para compilar o programa:

```
go build mock-producer.go
```

Será criado um arquivo executável, o `mock-producer` (ou `mock-producer.exe` para Windows). Esse programa utiliza variáveis de ambiente para configurar o acesso à fila. Você deverá criar as seguintes variáveis de ambiente (considerando que você está utilizando o usuário padrão do ActiveMQ):

- `QUEUE_HOST`, com o valor localhost;
- `QUEUE_PORT`, com o valor 61616;
- `QUEUE_USERNAME`, com o valor admin;
- `QUEUE_PASSWORD`, com o valor admin.

Se estiver usando Linux, pode criar no terminal assim:

```
export QUEUE_HOST=localhost
export QUEUE_PORT=61613
export QUEUE_USERNAME=admin
export QUEUE_PASSWORD=admin
```

Ou no Windows:

```
SET QUEUE_HOST=localhost
SET QUEUE_PORT=61613
SET QUEUE_USERNAME=admin
SET QUEUE_PASSWORD=admin
```

A porta 61613 é a porta padrão do protocolo STOMP (*Streaming Text Oriented Messaging Protocol*, ou Protocolo de Mensageria Orientado a Fluxo de Texto). Esse é um protocolo de mensagens

para softwares de intermediação (middlewares), como filas. A configuração de portas do ActiveMQ está no arquivo `activemq.xml` no diretório `conf` .

> IMPORTANTE! PRINCÍPIO DE BOA ARQUITETURA!
>
> Observe que o programa `mock-producer` é *configurável*. Ele possui valores predefinidos para a conexão, mas eles podem ser alterados externamente sem a necessidade de alterar o código-fonte do programa e recompilá-lo. A *configurabilidade* é uma característica desejável do software manutenível, aquele que precisa de manutenção.
>
> Os dados mutáveis devem ser separados do programa em arquivos de configuração ou variáveis de ambiente. Ao fazer isso, quando os dados mudam, precisamos apenas reiniciar o programa, evitando alterações no código-fonte, que podem inadvertidamente introduzir *bugs*. Então, quando escrever um programa, procure sempre deixá-lo *configurável*.

Com o ambiente devidamente configurado, você pode executar o programa `mock-producer` . Ele vai apresentar uma mensagem inicial enquanto estabelece a conexão com o ActiveMQ:

```
MOCK-PRODUCER: initializing data producing...
MOCK-PRODUCER: version 1.0.0
```

Assim que a conexão for estabelecida, o programa simulará que está lendo dados do Kubernetes e passará a enviar mensagens com supostos eventos de alteração de IP de pods. A mensagem consiste em um objeto no formato JSON, como no exemplo a seguir. O programa vai exibindo as mensagens que envia conforme as cria.

```
MOCK-PRODUCER: Has connectivity with localhost:61613
{"class": "audit","subclass":
"pod_ip","origin":"mock","dc":"mock","host":"mock","pod_namespace":"mock",
"pod":"mock","pod_ip":"mock","pod_ip_status":"mock","short_message":"mock/
```

```
mock:8","full_message":"mock/mock:mock:mock:2021-04-26 14:27:57.830195866
-0300 -03 m=+38.019616993","timestamp":"2021-04-26 14:27:57.830195866
-0300 -03 m=+38.019616993","logtype": "mock"}
```

Na página web de administração do ActiveMQ, você pode acompanhar as conexões do produtor com a fila. No menu horizontal, se você clicar sobre o item `Connections`, será levado para a página de conexões. Ela mostrará as conexões que o `mock-producer` está abrindo com o ActiveMQ, conforme exemplo da figura a seguir.

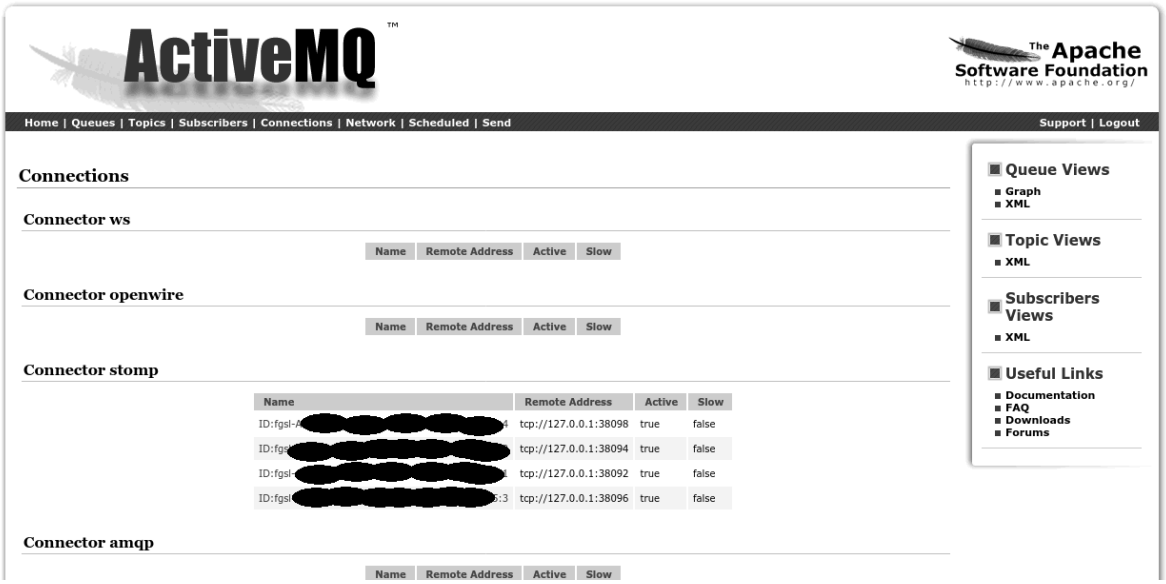

Figura 3.5: A página de conexões com a fila.

O programa `mock-producer`, uma vez iniciado, fica em *loop* e você pode interrompê-lo a qualquer momento com um CTRL+C. Você pode deixá-lo rodando por algum tempo e acompanhar, na página de filas, o crescimento do número de mensagens para verificar que o programa está realmente gravando na fila. Mas você pode interrompê-lo até que ativemos um outro programa, aquele que lerá os dados da fila, afinal a fila tem um limite de mensagens.

Não podemos apenas gravar, certo? Temos de ler. As mensagens não podem apenas entrar, elas têm de sair. O objetivo da fila é ser um canal intermediário entre dois sistemas, que não se comunicarão

de forma imediata. O produtor grava o dado em um momento e o consumidor lê o dado quando puder. A fila é um mecanismo de adiamento, como já falamos anteriormente.

O sistema consumidor

Agora vamos instalar um sistema que lê as mensagens da fila e simula seu envio para um servidor de logs. Esse sistema é o `mock-consumer`. Você pode instalá-lo usando o Git ou baixá-lo como um arquivo ZIP, como seu parceiro produtor.

Se você optar por instalar o `mock-consumer` com o Git, você executará o seguinte comando no terminal, em um diretório de sua escolha:

```
git clone https://github.com/fgsl/mock-consumer.git
```

Caso contrário, basta descompactar o arquivo ZIP, disponível no botão `Code` da página https://github.com/fgsl/mock-consumer, em um diretório de sua escolha.

Você terá como resultado, em qualquer uma das opções, a criação de um diretório `mock-consumer`. Dentro dele, há um programa escrito em linguagem Python, o `mock-consumer.py`. Para usá-lo, precisamos do Python. Você pode instalar o interpretador Python a partir desta página: https://www.python.org/downloads. Após instalar o interpretador Python, acesse via terminal o diretório `mock-consumer` e execute o seguinte comando para executar o programa:

```
python3 mock-consumer.py
```

Ele exibirá uma mensagem inicial, informando que vai aguardar para começar a leitura das mensagens:

```
Waiting 10 seconds to start...
```

Assim que iniciar a leitura, ele passará a exibir as mensagens lidas no formato JSON, como no exemplo que se segue:

```
MOCK-CONSUMER: received a message "{"class": "audit","subclass":
"pod_ip","origin":"mock","dc":"mock","host":"mock","pod_namespace":"mock",
```

"pod":"mock","pod_ip":"mock","pod_ip_status":"mock","short_message":"mock/mock:466","full_message":"mock/mock:mock:mock:mock:2021-04-26 14:35:36.863442698 -0300 -03 m=+497.052863844","timestamp":"2021-04-26 14:35:36.863442698 -0300 -03 m=+497.052863844","logtype": "mock"}"

> IMPORTANTE!
>
> O `mock-consumer` usa as mesmas variáveis de ambiente que o `mock-producer`. Se você estiver rodando os programas em sessões de terminal diferentes, terá de criar novamente as variáveis. A solução definitiva é criar essas variáveis globalmente para seu perfil de usuário. Assim você não terá de configurá-las cada vez que abrir uma sessão de terminal.

Agora você pode deixar os dois programas rodando por algum tempo e observar a página de filas. Como esses programas rodam muito rápido, a mensagem é tirada da fila logo que entra, assim o número de mensagens pendentes vai permanecer em zero. Você pode, como experimento, parar o `mock-consumer` após algum tempo para ver o indicador de mensagens não lidas aumentar rapidamente de valor.

Observe que ambos os programas imprimem as mensagens que, respectivamente, gravam e leem. Isso permite que você direcione a saída dos programas para arquivos de texto e compare o que foi gravado com o que foi lido. Ou seja, você pode testar se o que sai da fila é o mesmo que entra. E nós não terminamos nosso sistema. Nós só temos um microsserviço real até agora, que é a fila. O produtor e o consumidor são dublês: eles fingem que são os outros microsserviços. Isso é muito importante!

IMPORTANTE! PRINCÍPIO DE BOA ARQUITETURA!

Em um sistema baseado em componentes acoplados entre si, cada componente do sistema deve poder ser testado independente dos demais estarem prontos. Cada componente deve ser capaz de funcionar com dublês dos outros componentes, sem necessidade de utilizar os componentes reais. Isso permite testar se a comunicação entre componentes está seguindo o padrão estabelecido.

De forma análoga, cada microsserviço em um sistema baseado em microsserviços deve poder ser testado independente dos outros microsserviços existirem realmente. Você pode usar dublês de microsserviços, que rodem na mesma máquina do microsserviço que está sendo testado, mas que finjam estar em outras máquinas.

Você deve testar o sistema de forma completa, mas cada parte do sistema deve ser testada individualmente. Isso não vai garantir um sistema completamente confiável no teste geral do sistema, mas diminuirá a quantidade de falhas e assim reduzirá o tempo necessário para corrigi-las. As falhas no teste geral se limitarão a questões pontuais de integração entre todos os componentes, que não foram cobertas pelos testes individuais. Nós queremos realmente eliminar as falhas, mas, para fazer isso de forma produtiva, precisamos gradualmente reduzir a quantidade de falhas. É mais fácil tratar um conjunto limitado de falhas do que uma centena.

Bem, foi emocionante trabalhar com a fila, não? Ela funciona, mas o sistema não está pronto. Temos de substituir os dublês pelos atores reais. No próximo capítulo, substituiremos o dublê de produtor pelo produtor real. E prosseguiremos nossos comentários sobre princípios de arquitetura. Conforme prometido, vamos falar de teoria

de arquitetura apenas após mostrar algo prático. E deixamos uma frase para você refletir:

É melhor um princípio bem compreendido do que um conselho do Mestre dos Magos.

Capítulo 4
O microsserviço produtor

O que distingue uma época econômica de outra é menos o que se produziu do que a forma de o produzir. - Karl Marx.

O paradigma dos microsserviços permite que um sistema seja construído com diversos programas que se comunicam por mensagens em protocolos comuns, como HTTP, o que permite o uso de diversas linguagens de programação na construção desse sistema.

Poder usar qualquer linguagem de programação é como ter uma caixa de ferramentas com várias ferramentas. Você olha para o problema e procura na caixa aquela ferramenta mais adequada para resolvê-lo. Tenho que bater um prego? Uso um martelo. Tenho que apertar um parafuso? Uso uma chave de fenda, ou Philips. Tenho que cortar algum objeto? Uso uma serra.

O programador contemporâneo do mundo dos microsserviços deve ser um programador poliglota. Não no sentido de saber previamente todas as linguagens, embora deva conhecer algumas. Ele deve estar apto a aprender e implementar em qualquer linguagem, assim como um motorista deve estar apto a dirigir qualquer veículo. É claro que é desejável que o programador ou programadora busque se aprofundar em uma determinada linguagem para servir de conselheiro para outras pessoas desenvolvedoras.

Neste livro, mostraremos como organizar uma única aplicação implementada em várias linguagens. Sendo assim, cada parte do nosso sistema de auditoria utilizará uma linguagem diferente.

Se você estiver trabalhando com um software monolítico, em que todos os programas são executados em uma única máquina em um único ambiente, usar a mesma linguagem de programação faz todo

sentido. Mas no cenário dos microsserviços, não há essa limitação. Você não precisa construir o sistema todo com a mesma linguagem.

Estudaremos neste capítulo o programa que desempenha o papel de produtor real no nosso sistema de auditoria. O que o programa faz você já deve ter compreendido: ele captura dados do Kubernetes, os processa e os envia para uma fila.

Nós não paramos para estudar o código-fonte do dublê antes, porque esse código também aparece no programa real. O produtor real, assim como o dublê, é um programa escrito na linguagem Go, então entenderemos também o motivo pelo qual escolhemos um programa escrito nessa linguagem.

Antes de começarmos, vamos entender um pouco sobre a abstração do desenvolvimento de software, tópico importante para a configuração do podips-reader, como veremos adiante.

4.1 A abstração no desenvolvimento de software

As linguagens de programação têm diferentes camadas de abstração. Para facilitar o trabalho dos programadores e das programadoras, elas podem encapsular várias instruções do microprocessador em uma única instrução. É como substituir um longo discurso por uma única palavra.

A abstração é extremamente importante no desenvolvimento de software contemporâneo. Abstrair implica ignorar detalhes e se focar nos aspectos principais. Não podemos nos concentrar em tudo ao mesmo tempo, por isso estamos constantemente abstraindo. A abstração é um resumo ("resumo" em inglês é *abstract*).

A abstração no software significa organizar o código-fonte dos programas de modo que possamos compreender as instruções partindo do nível mais alto (mais genérico) até o mais baixo (mais específico).

Algumas linguagens abstraem mais o hardware do computador do que outras e assim tornam a pessoa programadora mais distante das operações reais da máquina. Outras linguagens abstraem menos e permitem inclusive intervenções pontuais diretas no hardware.

Cada projeto de linguagem de programação tem um objetivo, um público-alvo e um contexto histórico de criação. Há linguagens projetadas para cálculo, com pouca entrada e saída e muito processamento, que têm como público-alvo cientistas que trabalham em centros de pesquisa. Outras linguagens são projetadas para criar sistemas de informação, com muita entrada, muita saída, mas pouco processamento, e que têm como público-alvo analistas que trabalham em empresas comerciais.

4.2 A escolha pela linguagem Go

Kubernetes é um software criado pelo Google, que também criou a linguagem de programação Go. O Google produziu um software cliente em linguagem Go para dar acesso à API do Kubernetes. Isso evita que tenhamos de escrever muito código para poder interagir com o Kubernetes, além de ficar mais fácil utilizar o cliente Go usando a linguagem de programação Go.

É por isso que o programa que consome os dados do Kubernetes e grava na fila é escrito em Go. É uma decisão de conveniência, pela disponibilidade de um componente que faz a integração com um sistema externo.

A maneira mais rápida de desenvolver um software é não ter de desenvolvê-lo. Usar o que existe é mais rápido do que construir. Algumas decisões de adoção de componentes de software ocorrem por isso: é mais fácil e rápido usar o componente pronto do que escrevê-lo. Se o componente está disponível em uma determinada linguagem de programação, a tendência é que se crie um código em torno dele que use essa linguagem para manipulá-lo diretamente.

É importante explicar isso para que não fiquem dúvidas a respeito da escolha. A linguagem não foi escolhida porque gera um código que é executado mais rapidamente que o equivalente em outras linguagens, ou porque permite escrever menos código com suas instruções. O motivo foi a disponibilidade de um componente que evita ter de escrever mais código do que as regras de negócio do nosso sistema.

Tudo o que não for de nosso interesse direto, se possível, deve ser terceirizado para que nos concentremos no núcleo do problema que nos foi entregue. Procurar componentes que façam atividades necessárias, mas que não consistam nas tarefas particulares do sistema com o qual estamos trabalhando, faz parte da busca pela produtividade. Ser mais produtivo não consiste em ficar trabalhando mais tempo, mas sim em produzir mais no mesmo tempo. Queremos trabalhar menos e produzir mais.

Quando um sistema é monolítico, ficamos limitados à busca por componentes, porque tendemos a manter tudo escrito em uma única linguagem de programação uma vez que o sistema consiste em um único software. No paradigma dos microsserviços, essa limitação não existe. Cada microsserviço é livre para ser implementado na linguagem mais conveniente.

Havendo discutido as motivações da escolha da linguagem, que valem não somente para o microsserviço produtor mas para todos os demais, vamos analisar a implementação do microsserviço que lê os dados sobre mudanças de estado de pods do Kubernetes e

grava na fila. Esse microsserviço, o **podips-reader**, é o ator real que substitui o dublê **mock-producer** do capítulo anterior.

4.3 O podips-reader

Um programa Go é compilado. A partir do arquivo de código-fonte é gerado um arquivo com as instruções em linguagem de máquina. Esse arquivo, denominado **executável**, pode ser usado de forma independente do código-fonte. Ou seja, uma vez gerado o executável, não há necessidade do código-fonte para que o programa seja executado. O compilador Go e a documentação da linguagem podem ser obtidos em https://golang.org.

A linguagem de programação Go utiliza classes e objetos, mas não é uma linguagem orientada a objetos. Com ela, você pode criar classes ou apenas usar funções. Um programa Go é iniciado por uma função `main()`, assim vamos iniciar a análise do programa podips-reader por essa função.

O programa podips-reader é um software livre e aberto, e seu código-fonte está disponível em https://github.com/fgsl/podips-reader/blob/main/podips-reader.go.

O podips-reader é um programa sem interface gráfica. Ele roda em um laço de repetição sem fim, parando apenas com uma interrupção forçada - a ideia é que, uma vez iniciado, ele não pare. A saída do programa é feita diretamente no terminal.

Mesmo que um programa não tenha interação com o usuário (pelo menos um usuário humano), é necessário informar o que ele está fazendo para que a equipe de suporte tenha dados para trabalhar em caso de falha. É útil indicar as principais etapas do processamento na saída padrão do programa para poder seguir o rastro de sua execução. É assim que o programa **podips-reader** se inicia, com mensagens indicando que ele está começando:

```
func main() {
    fmt.Println("PODIPS-READER: initializing audit for Kubernetes Pods")
    fmt.Println("PODIPS-READER: version 1.1.0")
```

SINTAXE DO GO: ALGUNS DETALHES

Funções em Go começam com a palavra `func` .

Funções começam com letras minúsculas e nomes de métodos começam com letras maiúsculas.

O bloco de código em Go é igual ao da linguagem C, delimitado por chaves.

Go é uma linguagem de programação modular, assim como C. A modularização permite que os programas tenham o tamanho necessário quando compilados, porque você define exatamente o que vai usar. O inconveniente é ter de fazer as referências.

O objeto `fmt` só pode ser usado se for importado pela função `import()` , no início do arquivo, após a declaração `namespace` .

O operador de atribuição `:=` é usado para variáveis que não foram declaradas anteriormente. Para variáveis declaradas, usamos o operador `=` .

No capítulo anterior, falamos da *configurabilidade* como um aspecto desejável da arquitetura de um software. O podips-reader é configurável. A configuração de um programa, em geral, pode ser obtida por diversos meios, como leitura de arquivo, leitura de variável de ambiente ou leitura de API.

O comportamento do podips-reader pode ser modificado por parâmetros passados pela linha de comando, que é o caso da variável `options` , ou pela leitura de um arquivo de configuração, que é o caso da variável `config` :

```
options := getOptions()

config := getConfig(options)
```

A variável `options` é um mapa de dados, uma estrutura de dados que associa chaves a valores. Ela é obtida pela função `getOptions()`, que lê os parâmetros digitados no terminal:

```
// command-line arguments
func getOptions() map[string]bool {
    options := make(map[string]bool)
    var incluster *bool
    incluster = flag.Bool("incluster", true, "use in cluster, default is
true")
    flag.Parse()
    options["incluster"] = *incluster
    return options
}
```

Essa função tem poucas instruções. Poderíamos dispensar a função e colocar logo essas instruções dentro da função `main`, mas aí a função `main` ficaria maior e a leitura ficaria prejudicada. A modularização de software em funções ou procedimentos é uma questão de abstração, conceito extremamente importante como vimos anteriormente.

Ao colocar a leitura dos parâmetros passados pela linha de comando em uma função, reduzimos a compreensão dessa operação ao conhecimento de que ela existe. Se não estivermos interessados em modificar essa leitura, não precisamos sequer ler o código da função.

A manutenção de um software, seja por evolução, de uma forma tranquila, ou para suporte, de forma emergencial, exige que ignoremos tudo o que não for essencial para a solução do problema proposto. Abstrair nos ajuda a evitar distrações, a evitar a perda de tempo com a compreensão de algo que não é parte do problema.

A função `getOptions()` nos diz, apenas com seu nome, que estamos lendo opções. Se não quisermos saber como as opções são lidas ou não tivermos de alterar a forma de leitura das opções, passamos adiante em sua chamada, sem sermos obrigados a ver sua implementação.

> IMPORTANTE!
>
> A abstração ajuda a manter o foco.

Vamos abstrair a implementação da função `getConfig()`, chamada depois da `getOptions()`, porque ela não é relevante para nossa discussão. Precisamos apenas saber que ela recupera configuração para conexão com o Kubernetes.

Seguindo na sequência de instruções da função `main()`, encontramos um comando para dar uma pausa de 30 segundos:

```
time.Sleep(30 * time.Second)
```

O objeto `time` encapsula as instruções que ordenam ao processador esperar uma quantidade de segundos no método `Sleep`. Essa pausa obrigatória é para aguardar a configuração do ambiente do pod Kubernetes que contém o programa **podips-reader**, mais especificamente a aplicação das regras de acesso de rede que permitem que o programa se comunique com outros programas.

Essa pausa é como uma parada do carro no final de uma via secundária para olhar se a pista principal está livre antes de entrar.

Em seguida, temos a tentativa de criação do objeto cliente do cluster Kubernetes, com um tratamento de erro, que aborta o programa em caso de falha.

```
clientset, err := kubernetes.NewForConfig(config)
if err != nil {
```

```
    die("PODIPS-READER: ERROR: Can't create client configuration")
}
```

A maioria das linguagens de programação retorna apenas um valor como resultado de uma função ou método. A linguagem Go permite retornar mais de um valor. No caso do método `NewForConfig`, são retornados o objeto cliente do Kubernetes e o objeto de erro. Se o objeto de erro for igual a `nil`, não houve erro, então podemos prosseguir. Caso contrário, temos que tratar.

Confirmada a conexão com a API do Kubernetes, entramos em um laço de repetição sem fim definido, que tenta ler os registros de eventos do Kubernetes sobre pods em todos os namespaces. Usamos a API para obter um objeto `Watch`, que é um notificador de mudanças do Kubernetes.

É como se ficássemos perguntando continuamente: "mudou alguma coisa?". O intervalo entre cada pergunta é a soma do tempo gasto para formatar os dados e gravá-los na fila com um tempo fixo de dez segundos. Essa espera visa evitar que, sendo muito rápido, o laço pergunte sem que haja uma resposta positiva.

Caso ocorra uma falha na recuperação do objeto `Watch`, enviamos uma mensagem ao programa de monitoração, que será discutido no capítulo 6, e removemos o arquivo de status, o que indica ao Kubernetes que há uma falha com o **podips-reader**.

Se não houver falha, também enviamos uma mensagem ao monitor, mas com status de sucesso, e garantimos a existência do arquivo `kubernetes_status`. A ideia desse arquivo é dar uma indicação ao Kubernetes da necessidade de reiniciar o pod do programa, o que pode eventualmente restabelecer o funcionamento. Essa é a garantia de disponibilidade do podips-reader.

Finalmente, temos uma chamada a uma função `readMessagesFromDatabase()`, que tenta recuperar mensagens de mudança que não conseguiram ser gravadas na fila.

Em um ambiente distribuído, todos os nós podem falhar em algum momento. Por isso, é importante que cada nó tome medidas para evitar perda de dados e garantir a continuidade de um processo assim que as conexões forem restabelecidas.

O podips-reader tenta gravar as mensagens na fila e, se não conseguir, grava em um banco de dados. Assim que a fila se torna acessível novamente, ele envia as mensagens guardadas, de modo que nenhuma se perde. O efeito colateral é o envio fora de ordem, mas elas podem ser ordenadas no destino.

Chamando a função principal

A seguir temos o trecho de código do laço de repetição da função `main()`. Como última observação, temos a criação e remoção do arquivo de status feita pelo objeto `os`, que encapsula chamadas a comandos do sistema operacional.

```
    // continue even some error occurs
    for {
        // get pods in all the namespaces by omitting namespace
        // or specify namespace to get pods in particular namespace
        watch, err :=
clientset.CoreV1().Pods("").Watch(metav1.ListOptions{})
        if err == nil {
            http.DefaultTransport.(*http.Transport).TLSClientConfig =
&tls.Config{InsecureSkipVerify: true}
            resp, err := http.Get(getPodipsHost() +
"/kubernetes/200/success")
            _ = resp
            if err != nil {
                fmt.Println("PODIPS-READER WATCH: WARNING: ", err.Error())
            }
            os.Create("/tmp/kubernetes_status")
            listEvents(watch)
        } else {
            fmt.Println("PODIPS-READER WATCH: WARNING: ", err.Error())
            http.DefaultTransport.(*http.Transport).TLSClientConfig =
&tls.Config{InsecureSkipVerify: true}
```

```
            resp, err := http.Get(getPodipsHost() +
"/kubernetes/500/fail")
            _ = resp
            if err != nil {
                fmt.Println("PODIPS-READER WATCH: WARNING: ", err.Error())
            }
            os.Remove("/tmp/kubernetes_status")
        }
        readMessagesFromDatabase()

        time.Sleep(10 * time.Second)
    }
```

Mesmo que você não saiba exatamente o que cada objeto faz por desconhecimento dos componentes, a estrutura da programação Orientada a Objetos separa o ator da ação, o que ajuda a identificar quem faz o quê. Você sabe que o nome antes do ponto é o objeto, e que o nome depois do ponto é um método se este último tem parênteses à sua direita (*objeto.metodo()*), ou é um atributo se não tem parênteses (*objeto.atributo*). A Orientação a Objetos ajuda a identificar o fluxo de ações e os responsáveis por elas, abstraindo a forma como as ações são executadas.

Um detalhe que é apontado por Steve McConnell como importante na construção de código-fonte é a nomeação adequada de funções, classes e métodos. Um nome adequado ajuda o leitor ou leitora a entender o que aquilo faz, ou pelo menos com qual assunto aquilo está relacionado.

Por exemplo, o objeto `http` diz claramente com seu nome que opera com o protocolo HTTP. O método `Get()` desse objeto informa que ele está fazendo uma requisição do tipo GET. Desde que a pessoa que lê o código conheça os protocolos, compreenderá em alto nível o que está acontecendo. E se não conhecer, poderá usar os nomes como palavras-chave de pesquisa.

Não podemos ignorar que o desenvolvimento de software exige estudo contínuo e pesquisa. Nunca saberemos tudo o que

precisamos porque a tecnologia está sempre evoluindo e porque há muito conhecimento anterior que não precisamos usar de forma imediata. Então sempre precisamos consultar alguma fonte de referência. Se temos as palavras-chave adequadas, fica mais fácil encontrar.

Chamando as demais funções

Vamos seguir na análise do programa podips-reader, comentando a função `listEvents()`, que prepara a lista de eventos para ser enviada como uma mensagem para a fila caso a conexão com o Kubernetes seja estabelecida.

O objeto `Watch`, que é passado como argumento para a função `ListEvents()`, pode retornar uma coleção de eventos ocorridos no Kubernetes por meio do método `ResultChan()`. A função `ListEvents()` itera sobre essa coleção, capturando e formatando os dados de mudança de pods para dentro de um objeto do tipo `PodInfo`. O tipo `PodInfo` não é um tipo nativo da linguagem Go e sim um tipo composto definido pelo programador ou programadora.

Toda linguagem de programação possui tipos de dados que são associados às variáveis. Isso é necessário para identificar o que você deseja armazenar na memória de modo a alocar o espaço necessário e controlar as operações que podem ser feitas com os valores contidos nas variáveis.

Às vezes precisamos guardar diversos valores que estão relacionados diretamente entre si. Esse é o caso dos atributos dos pods alterados. Temos vários dados que são do tipo texto, um que é um número inteiro e um objeto.

Poderíamos lidar com eles por meio de variáveis independentes, mas não ficaria explícito para quem está lendo que eles têm de ser tratados em conjunto. Além disso, é mais legível enviar uma única

variável de um tipo composto como parâmetro em uma função ou método do que onze variáveis diferentes. Isso também é uma forma de abstração.

O tipo `PodInfo` é definido antes da declaração da função `main()` e reúne como atributo o conjunto de variáveis que precisamos manipular em vários pontos do programa de forma integrada:

```
type PodInfo struct {
    eventType watch.EventType
    exitCode int
    phase string
    objectIP string
    objectName string
    objectNamespace string
    podKind string
    podStatus string
    sendLog bool
    state string
    terminated string
}
```

Ao usar o tipo composto `PodInfo`, evitamos ter de escrever todas as variáveis que ele contém sempre que precisamos lidar com elas. Um tipo composto torna o programa mais conciso. Havendo esclarecido esse ponto, podemos prosseguir com a explicação da função `listEvents()`.

Já dissemos que ela itera sobre uma coleção de eventos. Essa iteração é feita pela estrutura `for`, que lê cada elemento da coleção e joga para a variável `event`. A partir do objeto inserido em `event`, um objeto do tipo `PodInfo` é preenchido.

Uma função `getPodStatusAndSendLog()` faz o trabalho de identificar nesse objeto `PodInfo` se ele deve ser enviado para a fila (porque alguns eventos não são relevantes). Se o objeto deve ser enviado, usamos a função `getQueue()` para retornar um objeto de conexão com a fila e enviamos uma versão em texto do objeto do tipo `PodInfo`, criada com a função `getDataForLog()`.

Se ocorrer uma falha no envio, a função `saveMessageToDatabase()` armazena os dados em um banco para tentar enviar mais tarde e remove o arquivo que registra o status de envio para a fila. Independente do sucesso ou fracasso no envio, o programa de monitoração é notificado por meio de uma mensagem HTTP (enviada pelo objeto `http`).

Enfim, podemos ver a seguir o código-fonte da função `listEvents()`.

```
func listEvents(w watch.Interface) {
    var pods = make(map[string]string)
    var data string
    var podInfo PodInfo
    var queue *stomp.Conn
    var err error

    for event := range w.ResultChan() {
        pod := event.Object.(*apiv1.Pod)
        podInfo.eventType = event.Type
        podInfo.objectIP = pod.Status.PodIP
        podInfo.objectName = pod.ObjectMeta.Name
        podInfo.objectNamespace = pod.ObjectMeta.Namespace
        podInfo.phase = fmt.Sprintf("%#v", pod.Status.Phase)
        podInfo = getPodStateTerminatedAndKind(podInfo, pod)

        index := podInfo.objectNamespace + "/" + podInfo.objectName

        podInfo = getPodStatusAndSendLog(podInfo, pods, index)

        if podInfo.sendLog {
            data = getDataForLog(podInfo)

            queue, err = getQueue()
            err = queue.Send(
                "/queue/pods",// destination
                 "application/json",// content-type
                []byte(data))// body
            if err != nil {
                fmt.Println("ERROR WHEN SENDING TO QUEUE " + err.Error())
                http.DefaultTransport.(*http.Transport).TLSClientConfig =
```

```
&tls.Config{InsecureSkipVerify: true}
                resp, err := http.Get(getPodipsHost() +
"/queue/write/500/fail")
                _ = resp
                if err != nil {
                    fmt.Println("PODIPS-READER MONITOR: WARNING: ",
err.Error())
                }
                os.Remove("/tmp/queue_status")
                saveMessageToDatabase(data)
            } else {
                http.DefaultTransport.(*http.Transport).TLSClientConfig =
&tls.Config{InsecureSkipVerify: true}
                resp, err := http.Get(getPodipsHost() +
"/queue/write/200/success")
                _ = resp
                if err != nil {
                    fmt.Println("PODIPS-READER MONITOR: WARNING: ",
err.Error())
                }
                os.Create("/tmp/queue_status")
            }
            queue.Disconnect()
        }

        msg := "(send? " + strconv.FormatBool(podInfo.sendLog) + ")" +
data

        fmt.Println(msg)
    }
}
```

Não é nossa intenção fazer um curso de programação Go neste livro. Abordamos o que foi necessário para a análise do programa podips-reader em até dois níveis de chamada (quer dizer, a chamada da função principal e depois a chamada de funções a partir da função principal).

Nosso objetivo aqui foi utilizar o programa **podips-reader**, que assume o papel do dublê **mock-producer**, para comentar aspectos

arquiteturais de um software que produz dados para uma fila.

O código-fonte do programa podips-reader está disponível para ser livremente estudado, copiado e modificado. Se você tiver um cluster Kubernetes em um provedor de nuvem, pode utilizá-lo. Basta configurar as variáveis exigidas na função `getConfig()`.

Prosseguiremos nossa análise arquitetural no próximo capítulo, abordando o programa que consome dados da fila.

Capítulo 5
O microsserviço consumidor

O consumo é a única finalidade e o único propósito de toda produção. - Adam Smith

No capítulo anterior, o ator principal da cena de produção de dados de eventos para a fila tomou o lugar de seu dublê. Agora é a hora de o ator principal da cena de consumo desses dados, o **podips-writer**, substituir seu dublê, o **mock-consumer**.

Diferente do dublê, que apenas exibe o que está recebendo, o programa podips-writer envia os dados de alteração de pods ao destino final, o sistema especialista em armazenamento de logs. O podips-writer é um programa escrito em Python, assim como seu dublê.

Justificamos a escolha da linguagem Go anteriormente pela disponibilidade de um componente que facilita a integração com o Kubernetes, a fonte de nossa matéria-prima. Bem, nossa justificativa para a escolha de Python é a disponibilidade de um componente que facilita a integração com o destino de nosso produto acabado, o **Fluentd**.

Fluentd é descrito como um software livre e aberto para unificar dados de log coletados de várias fontes. Ele está disponível em https://www.fluentd.org. Fluentd é usado por grandes empresas de tecnologia da informação e tem integração com diversos sistemas e protocolos geradores de logs.

Existem bibliotecas de classes para integração com Fluentd em diversas linguagens, incluindo Python, uma linguagem livre e aberta, cujo interpretador e documentação podem ser obtidos em https://www.python.org.

5.1 A escolha pela linguagem Python

Se a integração com Fluentd é possível em diversas linguagens, por que então foi escolhida a linguagem Python? O motivo real é que havia um programa legado em Python, que se comunicava com o Fluentd, e ele foi aproveitado. Um programa legado é simplesmente um programa que já existe, funciona e é mantido, porque não há um motivo forte o bastante para substituí-lo.

O mundo do software faz parte do mundo real, onde a inércia atua. Nada muda de movimento até que uma força o obrigue a fazê-lo. Às vezes queremos mudar alguma coisa, pois vemos que é possível fazer algo melhor, mas nem sempre temos a oportunidade. Se não há um retorno que justifique a mudança, a tendência é manter o que existe. Ou, falando de outra forma, se o cliente não paga para mudar, a empresa não se sente compelida a fazer isso.

Às vezes, é preciso mudar para não perder um cliente ou para conquistar novos. Isso é investimento e tem riscos. Exige estudo de mercado, não pode ser uma ação aleatória. A inércia mantém uma empresa em uma zona de conforto até que uma perda a force a mudar. O problema é que esperar pela perda pode fazer com que a mudança ocorra tarde demais, pois os concorrentes podem mudar antes. Então é bom tentar equilibrar a estabilidade dos produtos que já existem com manutenção das despesas atuais, inovação e fazendo algum investimento periodicamente. A inovação é importante, mas é difícil de ser feita, conforme já afirmou o economista austríaco Joseph Schumpeter.

É recomendável fazer melhorias contínuas no software a cada mudança realizada. Se pequenas mudanças forem feitas de forma frequente, o software evoluirá gradualmente. Simultaneamente, pode-se manter um fundo de investimento para inovação, que seja aplicado quando houver resultados de alguma pesquisa que indique a viabilidade de um novo produto (ou uma nova implementação). Essa é uma mudança por oportunidade.

A janela da oportunidade, segundo o professor Alexandre Graeml da UTFPR, abre-se poucas vezes, então é preciso aproveitar o momento certo para pular por ela. Depois que ela se fecha, a mudança só ocorre de forma emergencial, obrigando você a tomar decisões rapidamente sem poder ponderar sobre as opções. Mudar aproveitando uma oportunidade é como evitar um incêndio. Mudar por pressão de um evento que obriga uma ação imediata é como tentar apagar um incêndio em andamento.

A mudança no software depende das escolhas que fazemos para sua construção. O software contemporâneo é como uma empresa que tem diversos fornecedores. E uma empresa não escolhe seus fornecedores de qualquer jeito, certo? Ela deve estabelecer critérios.

Ao escolher um componente para terceirizar uma tarefa do seu software, o arquiteto precisa verificar se esse componente é atualizado com frequência e se há uma comunidade de usuários que produz uma documentação adicional àquela disponibilizada pelo fornecedor. Esses critérios indicam que o componente está em constante manutenção e evolução e que há pessoas às quais se pode recorrer para resolver dúvidas sem necessariamente acionar o fornecedor.

Precisamos de fornecedores, mas não queremos depender totalmente deles. A relação de parceria é desejável, a de dependência não. A diversidade de fornecedores ajuda a não criar uma dependência muito forte de uma única organização e reduz os riscos de abandono do componente. Como assim?

Veja, se um fornecedor resolve abandonar alguns componentes que fabrica e esses componentes são justamente todos os que você utiliza em seu sistema de software, você estará completamente abandonado. Mas se usar apenas alguns, o risco de abandono será menor. De qualquer forma, você precisará procurar outro fornecedor, mas reduzirá sua preocupação. Na pior das hipóteses, você pode ter de manter esse componente por conta própria (se ele for aberto, é claro).

Todas as linguagens de programação amplamente utilizadas têm repositórios de componentes livres e abertos para uso. Nem sempre você encontrará tudo o que precisa de forma aberta e terá eventualmente de comprar um componente fechado.

Os componentes fechados têm o risco maior de abandono, pois, se os fornecedores não mantêm mais o desenvolvimento deles, você não pode manter, pois não tem acesso ao código-fonte. E acesso ao código-fonte significa poder baixar o código-fonte para compilar ou interpretar em uma máquina de sua propriedade. Não adianta ver o código-fonte em uma sala de acesso restrito dentro de um shopping em que o fornecedor tenha uma loja.

No caso do software de código fechado, a necessidade de seguir padrões abertos é muito mais importante do que no código aberto. Se o fornecedor abandonar o componente, você pode encontrar outro que o substitua baseando-se nos padrões, que garantem a compatibilidade de comunicação. No pior dos casos, conhecendo os padrões, você pode criar o seu próprio componente.

Agora que fizemos essa reflexão, vamos analisar a implementação do programa consumidor da fila, o **podips-writer**, escrito, assim como seu dublê, em Python.

5.2 O podips-writer

No clássico livro *Alice no País da Maravilhas*, do matemático inglês Lewis Carroll, há um momento no julgamento do Valete de Copas em que o Coelho Branco pergunta ao Rei de Copas onde deve começar a ler uma carta. O Rei diz, sem titubear: "Comece pelo começo [...] e siga até o fim: daí pare".

No programa em Go, assim como ocorre na linguagem C e em Java, o início ocorre em uma função ou método com o nome `main()`.

Em Python não existe isso. O programa Python inicia com a primeira instrução localizada fora de uma função ou classe.

Python não exige um caractere, como ponto e vírgula, para indicar o final das instruções, assim como Go. Mas diferente desta, Python obriga a programadora a indentar seu código. Indentar significa inserir recuos nas linhas, indicando a hierarquia das instruções. Quanto maior o recuo, mais baixo o nível hierárquico da instrução. A indentação é importante para a legibilidade do código-fonte, pois mostra quais instruções estão subordinadas a outras dentro de blocos de código.

Linguagens de programação, como C, Java e PHP não obrigam a pessoa programadora a indentar, permitindo até que todas as instruções de um programa possam ser escritas em uma única linha. Python força quem programa a criar um código legível ao tornar a indentação uma regra de sintaxe.

Precisamos que a linguagem obrigue a indentação para que ela ocorra? Não necessariamente. Os ambientes integrados de desenvolvimento formatam arquivos de código-fonte indentando código sem necessidade do programador ou programadora.

Alguns ambientes, como o Eclipse, inclusive permitem programar gatilhos para que a formatação seja aplicada na hora de salvar o arquivo, garantindo que ele sempre esteja formatado de acordo com uma determinada convenção de código (que inclui a indentação).

Assim como fizemos no programa podips-reader, você verá que esperaremos dez segundos para começar a ler as mensagens da fila, para que as configurações de ambiente tenham tempo de serem aplicadas.

Avisaremos ao usuário do programa que estamos esperando, para indicar que o programa já começou. Desabilitaremos mensagens de advertência, pois já esperamos algumas exceções. Mensagens são boas para encontrar causas de problemas, desde que sejam claras

e não apareçam em excesso. Gerar muitas mensagens é como não gerar nenhuma.

Eis a seguir o código inicial do programa, que desabilita as mensagens e faz uma pausa de dez segundos:

```
warnings.filterwarnings("ignore")
print("Waiting 10 seconds to start...")
time.sleep(10)
```

Depois da pausa, entramos em um laço de repetição, para ficar continuamente lendo mensagens da fila. Uma função `getQueue()` recupera um objeto que encapsula a conexão com a fila. Aguardamos 60 segundos para que seja feita a leitura de mensagens e fechamos a conexão. Caso ocorra um erro, exibimos a mensagem de exceção, mas seguimos na tentativa de leitura.

É como se estivéssemos correndo e nos levantássemos rapidamente em caso de uma queda. Agimos como se nada tivesse acontecido. O programa não parará mesmo que haja exceções em sequência. Isso garante que ele esteja sempre disponível, mas não garante que ele vai executar sua função principal, que é ler as mensagens da fila e enviar para o Fluentd.

É por isso que precisamos monitorar o programa, mas isso será assunto do próximo capítulo. A seguir, temos a visão do bloco de código principal do programa podips-writer:

```
while True:
    try:
        queue = getQueue();
        print("Waiting for messages...")
        time.sleep(60)
        queue.disconnect()
    except Exception as e:
        print("ERROR:",e)
```

Você pode achar estranho esse código. Afinal, criamos um objeto, que abre a conexão, esperamos por 60 segundos e fechamos a

conexão. Esperamos o quê? Um milagre? Cadê as instruções que leem as mensagens da fila?

O padrão Sujeito/Observador

Neste programa, usamos um padrão de projeto que faz parte do paradigma da programação orientada a eventos, o padrão **Sujeito/Observador**. Esse padrão consiste em conectar dois objetos de modo que o observador seja notificado quando o estado (conjunto de atributos) do sujeito mudar.

O observador também é chamado de ouvinte na literatura, em uma analogia a alguém que reage quando um "ruído" ocorre no sujeito. Esse padrão de projeto cria uma lógica em que certas ações só são executadas sob demanda, evitando testes frequentes e desnecessários.

Quem é o sujeito em nosso programa? É o objeto de conexão com a fila. E quem é o observador? É um objeto da classe `LoggerListener`, que definimos no mesmo arquivo do programa principal. É o sujeito que se conecta ao observador, porque ele é o agente e o observador, o reagente.

Antes do bloco de código, que faz a conexão entre sujeito e observador, temos uma série de testes de existência de variáveis de ambiente. Isso ocorre porque o programa permite a sobreposição da configuração lida do arquivo de configuração por variáveis de ambiente. A regra é ler do arquivo de configuração, mas se uma variável de ambiente tiver valor para a mesma variável lida do arquivo, a primeira tem preferência. Isso permite que, caso seja necessário, rapidamente alteraremos a configuração do ambiente de produção, sem necessidade de publicar uma nova versão do programa. A configuração por variáveis de ambiente torna o ambiente de produção ágil para ações de suporte e atualização que não envolvam um novo desenvolvimento. Ela permite mudar apenas o que realmente mudou.

IMPORTANTE! PRINCÍPIO DE BOA ARQUITETURA!

Não perca tempo publicando novas versões de software em que apenas a configuração muda. Dados de configuração não devem fazer parte do código-fonte, devem residir em arquivos separados que sequer devem fazer parte do controle de versão.

Uma nova versão de software publicada deve conter alguma alteração no código-fonte: correção de bug, melhoria da estrutura ou nova funcionalidade. Alteração de dados de configuração não se encaixa em nenhum desses itens. A alteração de dados de configuração deve ser como ligar ou desligar um interruptor: você não precisa fazer uma nova instalação elétrica para pressionar o botão.

Após o tratamento dos dados de configuração, temos a criação do objeto de conexão `queue`, uma instância da classe `Connection` do pacote `stomp`. Esse pacote implementa o envio e recebimento de mensagens no protocolo homônimo, do qual falamos no capítulo 3.

Após o estabelecimento de conexão com o método `connect()`, conectamos o sujeito `queue` com o observador `LoggerListener` usando o método `set_listener()` do primeiro. O observador será notificado quando houver uma nova mensagem na fila. Isso é feito pelo método `subscribe()` do objeto `queue`, que faz uma "assinatura" do serviço de fila.

Quando a fila recebe novas mensagens, ela avisa o objeto `queue`. Este, por sua vez, avisa a instância de `LoggerListener`. A seguir, temos o código-fonte da função `getQueue()` completa:

```
def getQueue():
    print("Loading queue configuration...")
    try:
        with open("queue.config.json", "r") as f:
            queue_config = json.loads(f.read())
    except IOError:
```

```
        queue_config = {"QUEUE_USERNAME" : "", "QUEUE_PASSWORD": "",
"QUEUE_HOST": "", "QUEUE_PORT": ""}

    print("ActiveMQ configuration loaded from JSON file...")

    if  os.getenv('QUEUE_USERNAME') is not None:
        queue_username = os.getenv('QUEUE_USERNAME')
    else:
        queue_username = queue_config['QUEUE_USERNAME']
    print("queue_username defined...")

    if  os.getenv('QUEUE_PASSWORD') is not None:
        queue_password = os.getenv('QUEUE_PASSWORD')
    else:
        queue_password = queue_config['QUEUE_PASSWORD']
    print("queue_password defined...")

    if  os.getenv('QUEUE_HOST') is not None:
        queue_host = os.getenv('QUEUE_HOST')
    else:
        queue_host = queue_config['QUEUE_HOST']
    print("queue_host defined...")

    if  os.getenv('QUEUE_PORT') is not None:
        queue_port = int(os.getenv('QUEUE_PORT'))
    else:
        queue_port = int(queue_config['QUEUE_PORT'])
    print("queue_port defined...")

    print("Trying to connect with queue using " + queue_host + ":" +
str(queue_port) + "...")
    queue = stomp.Connection([(queue_host,queue_port)])
    queue.connect(queue_username, queue_password, wait=True)
    print("Connected to queue using user " + queue_username + " in host "
+ queue_host + ":" + str(queue_port))
    queue.set_listener('logger',     LoggerListener())
    queue.subscribe('/queue/pods',1)
    print("Queue /queue/pods subscribed")
    return queue
```

SINTAXE DO PYTHON: ALGUNS DETALHES

Funções e classes em Python começam com a palavra `def` .

Diferente de Go, Python pode definir classes e pode reaproveitar código de classes pelo mecanismo da herança.

O bloco de código em Python começa com dois pontos no final da instrução e termina quando se encontra uma instrução com o mesmo nível de indentação.

Python é uma linguagem de programação multiparadigma. Você pode usar em um mesmo arquivo Python classes, funções e código monolítico fora de ambas as estruturas.

O objeto `stomp` só pode ser usado se for importado pelo comando `import` , no início do arquivo.

O tipo das variáveis em Python é definido dinamicamente, a partir da atribuição de valores. Não há declaração de variáveis. Uma vez criadas, elas são fortemente tipadas, o que significa que não é possível fazer operações incompatíveis com seu tipo, como tentar somar um texto com um número.

A classe `LoggerListener` , que desempenha o papel de observadora, estende a classe ou herda da classe `ConnectionListener` da biblioteca `stomp` . A herança é um mecanismo de reúso da programação Orientada a Objetos.

A herança evita que você tenha de copiar e colar código desordenadamente. Na prática, a herança também copia código, mas de forma controlada. A grande diferença entre a cópia feita manualmente e a cópia feita pela herança é que na segunda, se o código-fonte original for alterado, todas as cópias serão atualizadas.

Há um comercial de pneus da Pirelli que diz "potência não é nada sem controle". Isso quer dizer que não adianta ter um carro

extremamente veloz mas com pneus ruins, que podem fazer o motorista perder o controle do veículo em uma pista ruim.

Da mesma forma, não adianta haver um oceano de códigos-fonte disponíveis para uso se você não tem controle sobre eles. Por meio da herança, a programação Orientada a Objetos disciplina o reaproveitamento de código, permitindo que você continue copiando, mas encapsulando os "originais" dentro de classes.

A classe `LoggerListener` herda dois métodos (explicitamente) da classe `ConnectionListener`: `on_error` e `on_message`. O primeiro é executado quando ocorre um erro na tentativa de ler mensagens da fila. O segundo é executado quando uma mensagem é lida com sucesso.

Nós não invocamos esses métodos. Eles são invocados quando os eventos de erro e leitura ocorrem, ou seja, há uma programação orientada a eventos na implementação de `ConnectionListener`. Ao se inscrever (ou assinar) no serviço de fila, o objeto `Connection` recebe da fila um erro ou uma mensagem, quando um deles é produzido.

A classe `Connection` é o sujeito que avisa o observador `ConnectionListener` que ocorreu uma mudança de estado.

Se a leitura é feita com sucesso, é instanciado um objeto de log com o método `getLogger()`. Esse objeto tem um método `emit()`, que retorna `true` se consegue enviar a mensagem para o Fluentd e `false` se não consegue.

Caso não consiga enviar, um objeto de conexão com a fila é criado com o método `getQueue()` e enviamos a mensagem de volta para a fila, para que ela não se perca. Além disso, em caso de falha apagamos o arquivo `log_status`. Se esse arquivo não for criado nas próximas tentativas, isso é sinal de que há uma falha contínua de comunicação com o Fluentd.

Independente de sucesso ou falha, enviamos uma mensagem HTTP para o programa de monitoração usando um objeto `requests`, que

encapsula a comunicação nesse protocolo.

Não é nossa intenção fazer um curso de programação Python neste livro. Abordamos o que foi necessário para a análise do programa podips-writer, cobrindo quase todo o código-fonte, exceto as funções `getLogger()` e `getPodipsHost()`, que não são relevantes o bastante para comentarmos. A linguagem Python, assim como Go, possui documentação aberta disponível e livros específicos sobre ela para que você possa se aprofundar em suas características particulares.

O código-fonte do programa podips-writer está disponível para ser livremente estudado, copiado e modificado. Se você tiver um servidor com Fluent instalado, pode utilizá-lo. Basta configurar as variáveis exigidas nas funções `getLogger()`, `getQueue()` e `getPodipsHost()`.

Nosso objetivo aqui foi utilizar o programa **podips-writer**, que assume o papel do dublê **mock-consumer**, para comentar aspectos arquiteturais de um software que lê dados de uma fila. Prosseguiremos nossa análise arquitetural no próximo capítulo, abordando o programa que monitora tanto o produtor quanto o consumidor da fila.

Capítulo 6
O microsserviço de monitoramento

Aquilo que não puderes controlar, não ordenes. - Sócrates

Nos capítulos anteriores, vimos a implementação do núcleo funcional de nosso sistema por meio dos programas **podips-reader**, **podips-writer** e **ActiveMQ**. Esses três programas criam uma linha de produção de mensagens de auditoria que tem origem em um cluster Kubernetes e fim em um servidor de logs Fluentd. Como seria maravilhoso se nós pudéssemos apenas iniciar esses três programas e sair para tomar café, confiantes de que eles ficarão fazendo seu trabalho ininterruptamente, pois afinal não são humanos e não precisam descansar.

Mas a realidade não é assim. Programas de computador não são entidades invulneráveis. Embora sejam apenas conjuntos de instruções, eles são executados em computadores que são entidades físicas, sujeitas às ações de degradação do mundo físico. Se um cabo de rede for cortado, o software não conseguirá estabelecer comunicação com os computadores ligados por ele. Se o disco rígido perder a seção TOC, o software não consegue mais ler arquivos. Se há interrupção de energia, as instruções não podem ser carregadas para a memória, e portanto não podem ser executadas.

Os problemas vão acontecer, cedo ou tarde. Não é questão de *se* e sim de *quando*. Mas se nos prepararmos, podemos agir rapidamente para resolvê-los.

A monitoração é fundamental para garantir a disponibilidade de sistemas de informação baseados em software. Precisamos vigiar todas as partes que compõem o sistema para saber se elas estão funcionando. Temos de agir como um supervisor de uma linha de produção, que se move ao longo da planta de operações da fábrica

para verificar se todos estão trabalhando adequadamente. O supervisor verifica se alguém não está fazendo seu trabalho, se está fazendo sem medidas de segurança e providencia as correções necessárias.

E não há somente uma pessoa vigiando em uma fábrica. Algumas plantas industriais colocam o escritório em uma posição superior que permite a visão da área de operações, de modo que os executivos da área tática, ou às vezes da estratégica, possam acompanhar o trabalho de perto e detectar alguma anomalia. Vários olhos são importantes no trabalho de garantia da continuidade de processos produtivos.

É como ter vários guardas ao longo das muralhas de uma fortaleza. Você precisa vigiar todas as posições, pois não sabe de onde virá o ataque. Da mesma forma, você precisa acompanhar todos os processos de produção, pois não sabe qual falhará, nem em que momento isso ocorrerá.

A monitoração precisa ser acompanhada de observabilidade para ser eficaz. Você pode instalar várias câmeras ao longo de uma instalação industrial, mas se não há uma pessoa que possa acompanhar todas as imagens de uma sala, as câmeras estão apenas enfeitando o local, tentando intimidar as pessoas apenas por sua presença. Não adianta vigiar sem tomar providências quando um evento indesejado ocorre.

Na monitoração de software, você pode coletar inúmeros indicadores sobre o ambiente em que o sistema está sendo executado, mas se não tiver uma informação clara sobre o status, eles serão inúteis. Muita informação é igual a nenhuma. Você também pode ter muitos dados sem ter informação, se eles não forem relacionados nem sintetizados.

É por isso que salas ou centros de monitoração usam telas que sintetizam os indicadores de modo a facilitar a interpretação sobre

os eventos que ocorrem, mostrando à pessoa técnica que as utiliza se há algum problema ou não.

6.1 O podips-monitor

Para saber se o trio de programas que compõem o núcleo funcional de nosso sistema está trabalhando de maneira efetiva, vamos utilizar um quarto programa para monitorá-los, o **podips-monitor**.

Esse programa é uma aplicação web em PHP, construído com o auxílio de um framework. Ao contrário dos outros, em que não há uma interação com um usuário final, o programa de monitoração precisa de uma interface para que a pessoa técnica de suporte possa saber o que está acontecendo. Um programa que gera páginas HTML com conteúdo legível para seres humanos, acessível por um navegador, é conveniente para informar técnicos ou técnicas de monitoração.

PHP é uma linguagem de programação construída para operar sobre o protocolo HTTP, para produzir aplicações web. Embora também possa ser utilizada para criar scripts de automação para infraestrutura de TI, o foco do PHP é a web.

Segundo dados do site W3Techs de 2021 (o link está disponível ao final do parágrafo), PHP é utilizado em mais de 79% dos websites cuja linguagem de programação é conhecida. A pilha de softwares Linux + Apache + PHP + MySQL é um conjunto maduro de ferramentas para a construção de aplicações web, mas PHP não está limitado a elas.

PHP pode ser usado com outros servidores web, como Nginx e Swoole, e tem suporte aos principais sistemas gerenciadores de banco de dados do mercado. Link do site W3Techs: https://w3techs.com/technologies/details/pl-php

PHP é uma linguagem concisa, altamente produtiva, que permite construir rapidamente aplicações Web. Embora tenha sido criada originalmente para ser embutida em páginas HTML, é possível utilizá-la em uma arquitetura de separação de camadas, onde o código de processamento não se mistura com o código de apresentação.

Isso não somente ajuda na organização da aplicação como permite construção e manutenção paralela por parte de web designers, desenvolvedores e desenvolvedoras.

Um framework, usado para auxiliar na construção do podips-monitor e que tem como correspondente em português a palavra *arcabouço*, é usado para denominar conjuntos de estruturas genéricas para a construção de software.

Ao detectar que há conjuntos de instruções repetidas em um software, a pessoa que programa pode isolar essa repetição em uma função ou em uma classe. Com o passar do tempo, vários projetos de software podem produzir muitas classes reutilizáveis, que podem ser reunidas em bibliotecas, para serem reaproveitadas em projetos futuros.

As bibliotecas de funções e de classes servem para evitar retrabalho e para utilizar implementações que estão maduras. Dessa forma, você pode aperfeiçoar componentes existentes em vez de ficar reinventando a roda a cada projeto.

Framework é o nível mais alto de reúso em software, segundo Erich Gamma. É uma reunião de bibliotecas de classe, porque framework presume o uso de programação Orientada a Objetos, mas não somente isso. Frameworks propõem o uso de um padrão de projeto chamado *inversão de controle*: em vez de você chamar o código genérico do framework, é o framework que chama o seu código específico.

O framework assume o controle da aplicação, oferecendo como vantagem a execução de uma cadeia de serviços sem necessidade

de intervenção da pessoa programadora. É o chamado princípio de Hollywood: não nos ligue, nós ligamos para você.

A inversão de controle dos frameworks, aliada a seus componentes reutilizáveis, permite que nos concentremos na implementação das regras de negócio da aplicação e não em atividades transversais relacionadas a protocolos de comunicação e interação com o sistema operacional.

Os frameworks fazem com que a pessoa desenvolvedora tenha foco naquilo que realmente interessa para o cliente ou usuário da aplicação, delegando as tarefas genéricas para componentes especializados nelas, uma vez que eles já têm essa experiência de projetos anteriores, nos quais foram criados e utilizados.

Como estamos construindo uma aplicação baseada em microsserviços, a aplicação de monitoração também é um microsserviço. Para construir microsserviços não precisamos usar um framework full stack, projetado para grandes aplicações, geralmente monolíticas. Um framework desse tipo traz muitas coisas desnecessárias para um microsserviço, que é uma aplicação reduzida.

Por isso, o podips-monitor foi construído com um microframework, um framework projetado para microsserviços. O framework utilizado para o nosso programa de monitoração é o **Mezzio**, um software livre e aberto, que faz parte do projeto **Laminas**. O Laminas é o nome atual do produto conhecido como **Zend Framework**, que existe desde 2005 e tem evoluído constantemente junto com a linguagem PHP.

Embora os frameworks ajudem no desenvolvimento ao evitar que codifiquemos coisas que já foram implementadas, eles aumentam a complexidade da aplicação no sentido de que precisamos compreender como eles funcionam.

Por isso, diferente dos capítulos anteriores, em que apresentamos de forma direta o código dos programas, que eram constituídos

basicamente de um único arquivo, desta vez explicaremos primeiro o passo a passo de como criamos a estrutura do projeto podips-monitor para depois comentar a implementação atual, que está disponível livremente em https://github.com/fgsl/podips-monitor.

6.2 Criação do podips-monitor

O Mezzio provê um esqueleto de projeto, com uma estrutura de pastas e alguns arquivos parcialmente preenchidos. Ou seja, ele dá um ponto de partida para quem desenvolve ao definir uma organização para os elementos de uma aplicação web.

Você pode baixar esse esqueleto em https://github.com/mezzio/mezzio-skeleton, mas precisará definir uma série de dependências (componentes) para que esse esqueleto se torne uma aplicação web funcional.

A maneira mais produtiva de começar um projeto de microsserviço PHP com o Mezzio é usando seu assistente de criação, o **MezzioInstaller**. Esse assistente é um programa PHP executado com o **Composer**, o gerenciador de dependências da comunidade PHP. Se você não tem o Composer, pode baixá-lo da página https://getcomposer.org/download.

O Composer é um programa PHP e por isso exige o interpretador PHP instalado. O PHP tem diversas distribuições, incluindo pacotes para instalação pelos sistemas de gerenciamento de pacotes das diversas distribuições Linux, como o **apt-get** e o **yum**.

Você pode também criar um ambiente completo de desenvolvimento de PHP usando o **XAMPP**, um pacote de softwares que combina Apache, PHP e MySQL, disponível para Windows, Linux e MacOs em https://www.apachefriends.org/pt_br/index.html.

Uma vez instalados PHP e Composer, o podips-monitor foi criado pelo seguinte comando:

```
composer create-project mezzio/mezzio-skeleton podips-monitor
```

O comando `create-project` do Composer cria uma estrutura de projeto a partir de um *template*. Esse template é um projeto de software que precisa estar registrado no **packagist.org**, o repositório de pacotes do Composer, ou em um repositório informado no arquivo `composer.json` . Neste caso, o `mezzio-skeleton` é um projeto cadastrado no packagist.org.

Esse comando cria, no diretório atual, um subdiretório com o nome informado após o nome do template. Nesse subdiretório é criada a estrutura definida pelo template e em seguida são iniciados os scripts definidos no arquivo `composer.json` do template.

A primeira parte da saída que vemos é o download da estrutura de projeto, que é descompactada e renomeada:

```
Creating a "mezzio/mezzio-skeleton" project at "./podips-monitor"
Installing mezzio/mezzio-skeleton (3.8.0)
- Downloading mezzio/mezzio-skeleton (3.8.0)
- Installing mezzio/mezzio-skeleton (3.8.0): Extracting archive
Created project in ./podips-monitor
```

A próxima parte da saída já faz parte do script de instalação, que pergunta que tipo de instalação você deseja.

```
> MezzioInstaller\OptionalPackages::install
Setting up optional packages
Setup data and cache dir
Removing installer development dependencies
What type of installation would you like?
[1] Minimal (no default middleware, templates, or assets;
configuration only)
[2] Flat (flat source code structure; default selection)
[3] Modular (modular source code structure; recommended)
Make your selection (2):
```

A instalação `Minimal` é para fazer os scripts PHP mais simples usando um middleware. Segundo a especificação PSR-7 do PHP-FIG, "um componente de middleware é um componente individual que participa, geralmente junto a outros componentes de middleware, no processamento de uma solicitação de entrada e na criação de uma resposta resultante, conforme definido pelo PSR-7".

Na prática, para aplicações Web, um middleware é simplesmente um programa que recebe uma requisição HTTP e devolve uma resposta HTTP. O PHP-FIG padroniza as implementações de middlewares para PHP com a interface `Psr\Http\Server\MiddlewareInterface`. A instalação `Minimal` é ideal para aplicações que consistem em apenas um arquivo PHP (com a adição de um arquivo de configuração).

PHP-FIG

O PHP-FIG é o Grupo de Interoperabilidade de Frameworks PHP. É um grupo que reúne representantes de vários projetos PHP de código livre e aberto. Esse grupo foi criado para estabelecer padrões de desenvolvimento para aplicações PHP Orientadas a Objeto.

O PHP-FIG publica especificações, que compreendem tanto regras quanto interfaces para padronizar a comunicação entre componentes de aplicações PHP. Os membros do grupo se comprometem a usar os padrões estabelecidos e trabalham para promovê-los entre a comunidade PHP. Você pode conhecer todas as especificações do PHP-FIG no site www.php-fig.org.

A instalação `Flat` cria um projeto para aplicações que envolvam mais do que um único arquivo. Ela traz alguns componentes, sem permitir escolha por parte de quem está instalando.

O podips-monitor foi criado com a instalação `Modular`, que cria um projeto com estrutura MVC e permite que você escolha as

implementações das principais categorias de componentes.

Ao escolher essa instalação, um arquivo chamado `ConfigProvider.php` é copiado para o diretório de classes, `src/App/src`. Esse arquivo é responsável por reunir as configurações dos arquivos que compõem as camadas do MVC.

> **MVC - Padrão Modelo-Visão-Controlador**
>
> O padrão MVC é uma proposta de arquitetura para aplicações que dividem as responsabilidades em três papéis bem definidos.
>
> O **modelo** é a camada responsável por definir os dados e as regras de negócio.
>
> A **visão** é a camada responsável por exibir os dados para o usuário.
>
> O **controlador** é o intermediário entre o modelo e a visão. O controlador é responsável por receber as requisições do usuário, localizar as instâncias do modelo que as tratem e enviar os dados processados pelo modelo para a visão.

Na saída do Composer, o MezzioInstallller mostra que fez a cópia do arquivo `ConfigProvider.php`.

```
Make your selection (2): 3
- Copying src/App/src/ConfigProvider.php
```

O contêiner de injeção de dependências do podips-monitor

Em seguida, o MezzioInstaller apresenta as opções de implementação para a injeção de dependência. O **contêiner de injeção de dependência**, ou *DI container*, é um objeto responsável por criar outros objetos a partir de fábricas, que são classes que sabem como criar objetos a partir de configurações. Essas fábricas são implementações do padrão de projeto *Factory Method*.

Usamos fábricas para encapsular a complexidade de objetos que dependem de outros objetos para serem criados - eles precisam da *injeção de dependências*. O encapsulamento da criação de objetos que dependem de outros em fábricas facilita a mudança de classes, desde que as interfaces continuem as mesmas.

Para o podips-monitor foi escolhido o componente `laminas-servicemanager`, que é o contêiner de injeção de dependências padrão do framework Laminas. Veja no código a seguir a instalação desse componente.

```
Which container do you want to use for dependency injection?
[1] Aura.Di (supported by laminas)
[2] Pimple (supported by laminas)
[3] laminas-servicemanager (supported by laminas)
[4] Auryn
[5] Symfony DI Container
[6] PHP-DI
[7] chubbyphp-container
Make your selection or type a composer package name and version
(laminas-servicemanager (supported by laminas)):
```

Quando o componente contêiner é selecionado, um arquivo de configuração específico para ele, `container.php`, é criado no diretório `config`.

```
Make your selection or type a composer package name and version
(laminas-servicemanager (supported by laminas)): 3
- Adding package laminas/laminas-servicemanager (^3.4)
- Copying config/container.php
```

O roteador do podips-monitor

O passo seguinte é a escolha do **roteador**. O roteador é o componente que interpreta os URLs e direciona a requisição HTTP para uma classe específica. Esse roteamento é feito a partir de uma configuração que conecta padrões de endereço com nomes de classes.

O roteador esconde do usuário a informação sobre quem está tratando sua requisição, ao mesmo tempo em que apresenta um endereço legível com significado claro para ele.

Para o podips-monitor foi escolhido o componente `laminas-router`, que é o roteador padrão do framework Laminas.

```
Which router do you want to use?
[1] Aura.Router (supported by laminas)
[2] FastRoute (supported by laminas)
[3] laminas-router (supported by laminas)
Make your selection or type a composer package name and version
(FastRoute (supported by laminas)):
```

Quando o componente roteador é selecionado, um arquivo de configuração específico para ele, `routes.php`, é criado no diretório `config`.

```
- Adding package mezzio/mezzio-laminasrouter (^3.0.1)
- Whitelist package mezzio/mezzio-laminasrouter
- Copying config/routes.php
```

O mecanismo de template do podips-monitor

O próximo passo é a escolha do **mecanismo de template**, que é o componente responsável pela geração de páginas HTML. Esse componente é opcional, pois alguns microsserviços podem não precisar exibir páginas HTML, mas apenas expor dados em algum formato que possa ser consumido por outra aplicação, como JSON ou XML.

Para o podips-monitor foi escolhido o componente `laminas-view`, que é o mecanismo de template padrão do framework Laminas.

```
Which template engine do you want to use?
[1] Plates (supported by laminas)
[2] Twig (supported by laminas)
[3] laminas-view installs laminas-servicemanager (supported by
laminas)
[n] None of the above
```

```
Make your selection or type a composer package name and version
(n):
```

Quando o componente de templates é selecionado, alguns templates são copiados para um subdiretório específico. Esses templates definem páginas de erro, a página de layout e uma página inicial.

```
- Adding package mezzio/mezzio-laminasviewrenderer (^2.2)
- Whitelist package mezzio/mezzio-laminasviewrenderer
- Copying src/App/templates/error/404.phtml
- Copying src/App/templates/error/error.phtml
- Copying src/App/templates/layout/default.phtml
- Copying src/App/templates/app/home-page.phtml
```

O componente de manipulação de erros do podips-monitor

O próximo passo é a escolha do componente de manipulação de erros. Esse componente é específico para o desenvolvimento, pois ele tem o propósito de permitir a depuração da aplicação pelo desenvolvedor ou desenvolvedora.

O Mezzio só tem suporte atualmente para um componente, o **Whoops**. A outra opção é ficar sem esse componente, o que não compromete a funcionalidade da aplicação, mas deixa quem desenvolve sem uma ferramenta adicional para corrigir erros.

O Whoops foi selecionado para o podips-monitor. O código a seguir mostra como selecionar o Whoops para a instalação.

```
Which error handler do you want to use during development?
[1] Whoops (supported by laminas)
[n] None of the above
Make your selection or type a composer package name and version
(Whoops (supported by laminas)):
```

Após a escolha do Whoops, o MezzioInstaller copia o template de arquivo de configuração de desenvolvimento (`development.local.php.dist`). Isso encerra a execução do MezzioInstalller, que é removido em seguida.

```
- Adding package filp/whoops (^2.7.1)
- Copying config/autoload/development.local.php.dist
Remove installer
Removing composer.lock from .gitignore
Removing Mezzio installer classes, configuration, tests and docs
```

Instalação de dependências

Na sequência, é iniciada a instalação das dependências do projeto, que são os componentes definidos nas chaves `require` e `require-dev` do arquivo `composer.json`.

Os componentes do Laminas trazem fábricas para a criação de seus objetos, por isso, quando o primeiro componente começa a ser instalado, o script de instalação pergunta se a classe de configuração do componente deve ser injetada no arquivo de configuração global.

Ao responder afirmativamente, permitimos que o contêiner de injeção de dependências tenha acesso às fábricas do componente.

Para o podips-monitor, escolhemos injetar a classe `ConfigProvider` para todos os componentes. Para isso, basta escolher injetar o primeiro e depois dizer que quer lembrar dessa opção para outros pacotes de software.

```
Installing dependencies from lock file (including require-dev)
Please select which config file you wish to inject 'Laminas\
Validator\ConfigProvider' into:
[0] Do not inject
[1] config/config.php
Make your selection (default is 1):
Remember this option for other packages of the same type? (Y/n)Y
```

Após a instalação das dependências, que são baixadas da rede pelo Composer, são gerados os arquivos para carregamento das classes, que evitam que a aplicação use comandos `require` e `include`.

Também é configurado o componente `PHP CodeSniffer`, que faz verificação de estilo de codificação, e é habilitado o modo de desenvolvimento, que exibe uma barra de status na parte inferior das páginas web da aplicação, permitindo ao desenvolvedor ou desenvolvedora monitorar indicadores do ambiente sem ter de consultar outras ferramentas.

```
Generating autoload files
composer/package-versions-deprecated: Generating version class...
composer/package-versions-deprecated: ...done generating version
class
90 packages you are using are looking for funding.
Use the `composer fund` command to find out more!
PHP CodeSniffer Config installed_paths set to ../../laminas/laminas-
coding-standard/src,../../slevomat/coding-standard,../../
webimpress,../../webimpress/coding-standard/src
> laminas-development-mode enable
You are now in development mode.
```

Ao final do processo de instalação, a estrutura de pastas do podips-monitor ficou como mostra a figura a seguir:

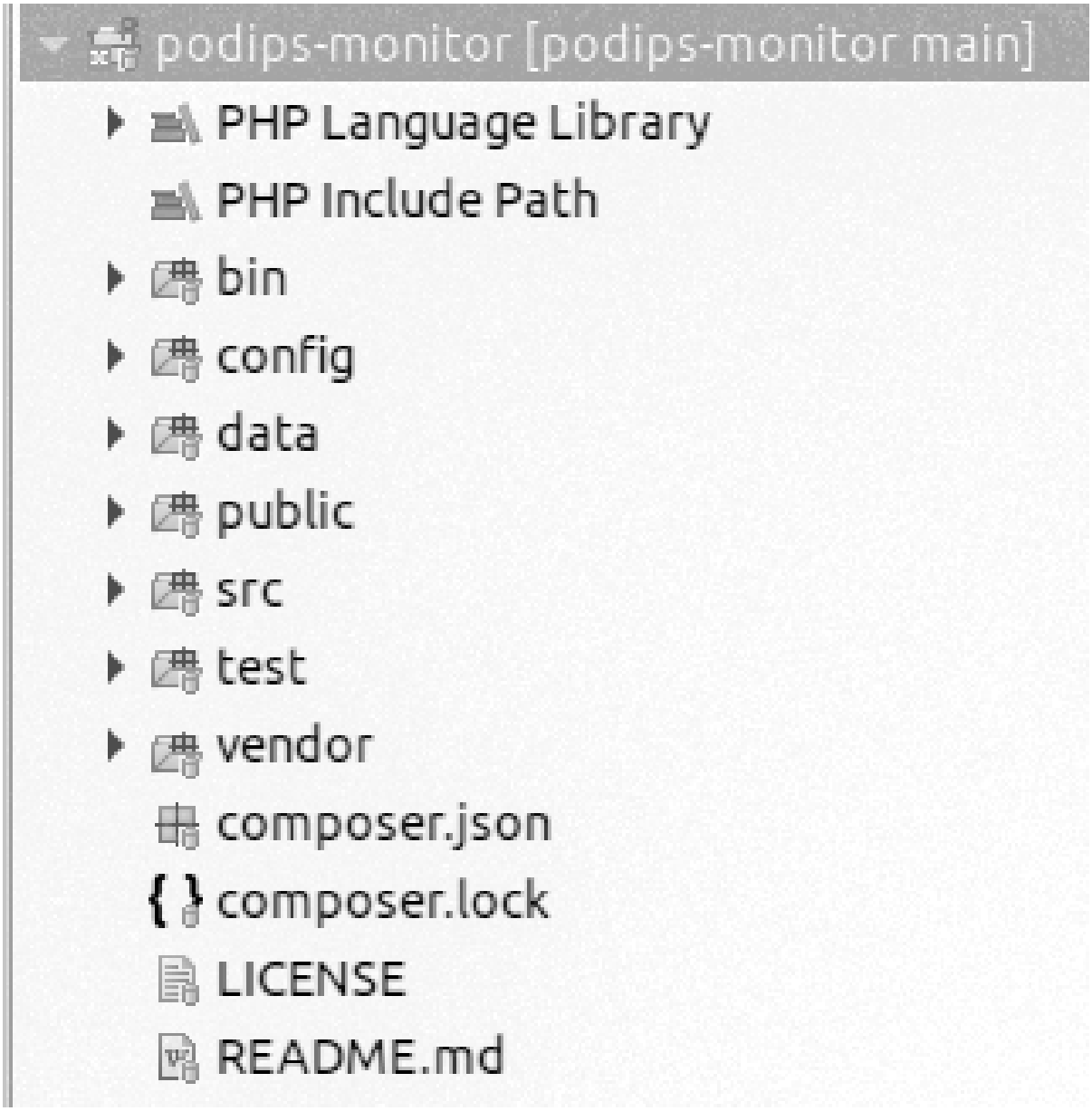

Figura 6.1: A estrutura de pastas do podips-monitor.

- A pasta **bin** contém o script para limpeza do cache, `clear-config-cache.php` .
- A pasta **config** contém os arquivos de configuração da aplicação.
- A pasta **data** serve para armazenar dados locais da aplicação. Na subpasta **cache**, fica um arquivo que combina todos os arquivos de configuração em um único.

- A pasta **public** contém o arquivo `index.php` , que é o ponto de partida da aplicação.
- A pasta **src** contém as classes e os templates de página da aplicação.
- A pasta **test** serve para conter os testes automatizados.
- A pasta **vendor** é criada pelo Composer e contém as dependências da aplicação, que são as bibliotecas de componentes utilizadas pela aplicação.

Uma vez criada a estrutura da aplicação, foi realizada a implementação das funcionalidades, constituindo o código-fonte disponível no Github. Na próxima seção, vamos falar sobre isso.

6.3 Implementação do podips-monitor

Vamos compreender o processo e as partes fundamentais do programa podips-monitor, uma aplicação implementada em PHP.

SINTAXE DO PHP: ALGUNS DETALHES

PHP é uma linguagem de programação cujo interpretador é implementado em linguagem C e sua sintaxe é similar em vários aspectos à linguagem C e em outros à linguagem C++.

Blocos de código em PHP são delimitados por chaves ({}).

As instruções em PHP terminam em ponto e vírgula (;). Você pode colocar várias instruções em uma mesma linha lógica de arquivo, o que importa para o interpretador para saber que a instrução terminou é o ponto e vírgula.

Funções em PHP são blocos de código com o nome `function`. Classes em PHP começam com a palavra `class`. Blocos de código com o nome `function` dentro de uma classe são tratados como métodos da classe.

PHP é uma linguagem de programação multiparadigma. Você pode, em um mesmo arquivo PHP, usar classes (paradigma orientado a objetos), funções (paradigma estruturado) e código monolítico (sem qualquer paradigma) fora de classes e funções.

PHP pode reaproveitar código de classes pelo mecanismo da herança e de blocos de código por meio de *traits*, que são unidades de reúso independentes de classe.

PHP foi projetado originalmente para ser enxertado em arquivos HTML, o que facilita o uso de instruções PHP para gerar conteúdo dinâmico em páginas web.

O tipo das variáveis em PHP é definido dinamicamente, a partir da atribuição de valores. Todas as variáveis em PHP começam com cifrão ($). Não há declaração de variáveis. Após serem criadas, elas são fracamente tipadas, o que significa que é possível fazer operações diferentes do tipo da variável, porque PHP tenta converter as variáveis de acordo com uma operação.

Isso quer dizer que, se você tentar somar uma variável do tipo **string** com uma variável do tipo **integer**, o PHP automaticamente tentará converter a variável **string** em um número porque a operação aritmética exige números como operandos.

Definindo as rotas do podips-monitor

Vamos começar a explicação sobre o funcionamento do programa podips-monitor pela parte que cuida da interpretação das requisições HTTP, que são o ponto de partida para um ciclo de processamento de dados em uma aplicação web.

O arquivo `routes.php`, na pasta `config`, contém as rotas da aplicação, que são as ligações entre URLs e as classes que processarão os URLs. A aplicação podips-monitor tem sete rotas:

- **home**: exibe a página inicial, com o painel de monitoração.
- **api**: exibe a documentação da API.
- **api.ping**: retorna um JSON indicando que a aplicação está ativa.
- **kubernetes**: recebe dados de status da leitura do Kubernetes.
- **queue**: recebe dados de status de leitura e escrita na fila.
- **log**: recebe dados de status de gravação no Fluentd.
- **podips**: exibe dados de status de uma operação.

O bloco de código a seguir mostra a definição dessas rotas, com o uso do método `get()` da classe `Application`. Nesse método, nós passamos como parâmetros o endereço relativo à aplicação, o nome da classe que o tratará e o nome da rota.

O método `get` permite que o endereço especificado seja requisitado com HTTP GET. O roteamento funciona como uma lista branca: apenas o que é definido é permitido. Se, por exemplo, alguém tentar enviar uma requisição HTTP POST para o endereço `/api`, receberá um erro HTTP 405.

```
return static function (Application $app, MiddlewareFactory $factory,
ContainerInterface $container): void {
    $app->get('/', App\Handler\HomePageHandler::class, 'home');
    $app->get('/api', App\Handler\ApiHandler::class, 'api');
    $app->get('/api/ping', App\Handler\PingHandler::class, 'api.ping');
    $app->get('/kubernetes/:code/:message',
App\Handler\KubernetesHandler::class, 'kubernetes');
    $app->get('/queue/:operation/:code/:message',
App\Handler\QueueHandler::class, 'queue');
    $app->get('/log/:code/:message', App\Handler\LogHandler::class,
'log');
    $app->get('/podips/:operation', App\Handler\PodipsHandler::class,
'podips');
};
```

O método handle e os dados de monitoração

Por convenção, todas as requisições são enviadas para o método `handle()` da classe associada à rota. As classes que processam as requisições em uma aplicação Mezzio são chamadas de *Handlers*. Essas classes implementam a interface `Psr\Http\Server\RequestHandlerInterface`, definida pelo PHP-FIG na especificação PSR-15.

As requisições HTTP são encapsuladas em objetos que implementam a interface `Psr\Http\Message\ServerRequestInterface`, definida também na PSR-15.

No Mezzio, as classes *Handler* representam a camada de controle no padrão MVC. Como um microsserviço, essas classes devem ser pequenas, com apenas um método público, o método `handle()`. Demais métodos devem ser apenas o construtor da classe e métodos privados para auxiliarem o método `handle()`. As classes *Handler* ficam na pasta `src/App/src/Handler`.

O método `handle()` da classe `App\Handler\HomePageHandler`, que exibe a *homepage*, cria quatro blocos de dados de monitoração: leitura do Kubernetes, escrita na fila, leitura da fila e escrita no Fluentd. Esses blocos indicam se a última execução da operação foi feita com

sucesso e se, em caso de falha, um e-mail de alerta foi enviado para o endereço cadastrado.

O método `handle()` deve retornar um objeto da interface `Psr\Http\Message\ResponseInterface` do PHP-FIG. Na classe `App\Handler\HomePageHandler`, esse método retorna um objeto da classe `Laminas\Diactoros\Response\HtmlResponse`, que implementa essa interface. O Laminas Diactoros é um componente para comunicação usando o protocolo HTTP.

Vamos entender passo a passo a implementação do método `handle()` de `App\Handler\HomePageHandler`. Primeiro, o método captura o host da aplicação, usando a variável superglobal `$_SERVER`.

```
$podipsHost = $_SERVER['REQUEST_URI'];
```

Depois, preenchemos as variáveis com dados sobre a operação de leitura do Kubernetes. Uma classe da camada de modelo, `App\Model\Monitor`, retorna um objeto JSON com o status da leitura. Esse objeto é usado para configurar a cor de exibição da mensagem por meio do método `getColor()` (vermelho para falha, laranja para alerta e verde para sucesso).

O método `getBlock()` retorna um bloco de texto com o status da leitura colorido de acordo com a variável `$color`. O método `sendMessage()` retorna o status de envio da mensagem de e-mail, de acordo com o status da leitura do Kubernetes. Esse status é concatenado ao bloco de texto, para informar se o e-mail foi enviado ou não.

```
$json = Monitor::getInstance()->getKubernetesReadingStatus();
$color = $this->getColor($json);
$kubernetesReadingStatusBlock = $this->getBlock($json, $color);
$kubernetesReadingSendMailStatus = $this->sendMessage((int)$json-
>code,'Fail to read Kubernetes', $kubernetesReadingStatusBlock . "
<p>$podipsHost</p>");
$kubernetesReadingStatusBlock .= "
<p>$kubernetesReadingSendMailStatus</p>";
```

Os três blocos que se seguem fazem os mesmos procedimentos, respectivamente, para as operações de escrita na fila, leitura da fila e escrita no Fluentd.

```
$json = Monitor::getInstance()->getQueueWritingStatus();
$color = $this->getColor($json);
$queueWritingStatusBlock = $this->getBlock($json, $color);
$queueWritingSendMailStatus = $this->sendMessage((int)$json->code, 'Fail
to write Queue', $queueWritingStatusBlock . "<p>$podipsHost</p>");
$queueWritingStatusBlock .= "<p>$queueWritingSendMailStatus</p>
<p>$podipsHost</p>";

$json = Monitor::getInstance()->getQueueReadingStatus();
$color = $this->getColor($json);
$queueReadingStatusBlock = $this->getBlock($json, $color);
$queueReadingSendMailStatus = $this->sendMessage((int)$json->code, 'Fail
to read Queue', $queueReadingStatusBlock . "<p>$podipsHost</p>");
$queueReadingStatusBlock .= "<p>$queueReadingSendMailStatus</p>";

$json = Monitor::getInstance()->getFluentdWritingStatus();
$color = $this->getColor($json);
$fluentdWritingStatusBlock = $this->getBlock($json, $color);
$fluentdWritingSendMailStatus = $this->sendMessage((int)$json->code, 'Fail
to write Fluentd', $fluentdWritingStatusBlock . "<p>$podipsHost</p>");
$fluentdWritingStatusBlock .= "<p>$fluentdWritingSendMailStatus</p>";
```

Os blocos de texto são reunidos em um array que é enviado como segundo argumento do método `render()` da instância de `Mezzio\Template\TemplateRendererInterface`. Esse método processa a página definida no primeiro argumento e retorna o conteúdo para o construtor da classe `Laminas\Diactoros\Response\HtmlResponse`. Essa classe cria uma resposta HTTP, que encerra o processamento da requisição, entregando o conteúdo para o browser do usuário.

As dependências do método handle

O método `handle()` de `App\Handler\HomePageHandler` depende de dois objetos para funcionar: uma instância de `Mezzio\Template\TemplateRendererInterface` e uma instância de

`App\Model\Mail`. Essas instâncias são recebidas no construtor da classe:

```
public function __construct(TemplateRendererInterface $template, Mail
$mail) {
    $this->template = $template;
    $this->mail     = $mail;
}
```

As classes *Handler* no Mezzio não são instanciadas diretamente, mas criadas por fábricas. Quem instancia a classe `App\Handler\HomePageHandler` é a classe `App\Handler\HomePageHandlerFactory`.

As classes fábricas no Mezzio são classes que implementam o método `__invoke()` do PHP. Esse método permite que um objeto seja invocado como se fosse uma função. As fábricas recebem uma instância do contêiner de injeção de dependência, a partir da qual qualquer outro objeto pode ser recuperado.

No método `__invoke()` da classe `App\Handler\HomePageHandlerFactory`, o contêiner é usado para recuperar uma instância de `Mezzio\Template\TemplateRendererInterface` e para recuperar a configuração de e-mail para criar um objeto da classe `App\Model\Mail`.

O Mezzio precisa saber qual fábrica será usada para criar cada *Handler*. Essa associação é feita na classe `ConfigProvider`, que fica na pasta `src/App/src`.

O método `getDependencies()` dessa classe define duas categorias de associações: `invokables`, que são as classes que não precisam de injeção de dependências, ou seja, não implementam o método construtor, e `factories`, que são as classes que precisam de injeção de dependências, ou seja, têm construtores definidos com parâmetros.

A seguir vemos o método `getDependencies()` da classe `ConfigProvider` no podips-monitor:

```
public function getDependencies(): array
{
    return [
        'invokables' => [
            Handler\PingHandler::class          =>
Handler\PingHandler::class,
            Handler\LogHandler::class           =>
Handler\LogHandler::class,
            Handler\KubernetesHandler::class    =>
Handler\KubernetesHandler::class,
            Handler\QueueHandler::class         =>
Handler\QueueHandler::class,
            Handler\PodipsHandler::class        =>
Handler\PodipsHandler::class,
        ],
        'factories'  => [
            Handler\HomePageHandler::class      =>
Handler\HomePageHandlerFactory::class,
            Handler\ApiHandler::class           =>
Handler\ApiHandlerFactory::class
        ],
    ];
}
```

Vamos voltar para o método `handle()` da classe `App\Handler\HomePageHandler`. Ele termina com o retorno de um objeto da classe `Laminas\Diactoros\Response\HtmlResponse`. Esse objeto recebe o conteúdo processado pelo método `render` da instância de `Mezzio\Template\TemplateRendererInterface`.

O método `render()` recebe como primeiro argumento um texto que indica qual é o diretório do template de página HTML e qual é o arquivo da página a ser processado. Esses dois dados são separados por um duplo dois-pontos (::).

```
return new HtmlResponse($this->template->render('app::home-page', $data));
```

A indicação do diretório não é um caminho absoluto, mas um apelido para esse caminho. Na classe `App\ConfigProvider`, há um método `getTemplate()`, que é usado para definir apelidos para os

diretórios de templates. Esse método informa ao objeto de template onde encontrar o arquivo. A extensão `.phtml` é assumida como padrão.

```
    public function getTemplates(): array
    {
        return [
            'paths' => [
                'app'    => [__DIR__ . '/../templates/app'],
                'api'    => [__DIR__ . '/../templates/api'],
                'error'  => [__DIR__ . '/../templates/error'],
                'layout' => [__DIR__ . '/../templates/layout'],
            ],
        ];
    }
```

Recuperando o conteúdo dos arquivos de status

Você viu anteriormente, na seção sobre o método `handle()` de `App\Handler\HomePageHandler`, que ele faz uso repetido da classe `App\Model\Monitor`. Essa classe é responsável por criar e recuperar o conteúdo dos arquivos de status.

No cabeçalho dessa classe, nós temos a definição dos nomes dos arquivos de status por meio de constantes.

Há dois arquivos que gravam os status gerais dos programas podips-reader e podips-writer:

```
    const PODIPS_WRITER_STATUS = 'podips-writer-status.json';
    const PODIPS_READER_STATUS = 'podips-reader-status.json';
```

E há um arquivo para cada operação do processo geral do sistema podips:

```
    const LAST_KUBERNETES_READING = 'last-kubernetes-reading.json';
    const LAST_QUEUE_WRITING = 'last-queue-writing.json';
    const LAST_QUEUE_READING = 'last-queue-reading.json';
    const LAST_LOG_WRITING = 'last-log-writing.json';
```

Uma outra constante define o diretório onde esses arquivos são gravados:

```
const DATA_FOLDER = APP_ROOT . DIRECTORY_SEPARATOR . 'data' .
DIRECTORY_SEPARATOR . 'files' . DIRECTORY_SEPARATOR;
```

PHP permite que a pessoa desenvolvedora escreva código-fonte conciso, graças a uma grande abstração de funcionalidades encapsuladas em funções.

A operação do sistema de arquivos pode ser feita em PHP por meio de diversas funções, como `file_exists()`, que verifica se um arquivo existe; `file_get_contents()`, que lê o conteúdo de um arquivo (transferindo para uma variável); e `file_put_contents()`, que grava o conteúdo de uma variável em um arquivo. Essas três funções são usadas no método privado `getJsonFile()` da classe `App\Model\Monitor`, que retorna o arquivo de status solicitado, criando-o se ele não existir.

```
    private function getJsonFile($path)
    {
        $datetime = date(\Datetime::ISO8601);
        try {
            if (!file_exists($path)){
                $json = <<<JSON
{
    "code" : "0",
    "message" : "no message",
    "datetime": "$datetime",
    "remoteaddr" : "unknown",
    "remotehost" : "unknown"
}
JSON;
                file_put_contents($path, $json);
            }
            $json = json_decode(file_get_contents($path));
        } catch (\Exception $e) {
            $jsonError = '{"code": "0","message": "error","datetime" :
"error", "remoteaddr": "error", "remotehost": error"}';
            $json = json_decode($jsonError);
```

```
        }
        return $json;
    }
```

Observe no método anterior que há um bloco de texto delimitado por um identificador, que segue os caracteres `<<<` , e termina com esse mesmo identificador. Neste caso, o identificador é a palavra JSON. Essa sintaxe, chamada **heredoc**, permite escrever textos da forma como queremos que eles sejam gravados, com quebra de linha e tabulação. Isso torna os blocos de texto em programas PHP mais legíveis.

Limitando a criação de instâncias

A classe modelo `App\Model\Monitor` é usada pelo controlador `App\Handler\HomePageHandler` . Em PHP, objetos são criados pelo operador **new**, mas não vemos esse operador no método `handle()` do controlador. Por quê? Porque a classe `App\Model\Monitor` implementa o padrão de projeto *Singleton*, que limita a criação de objetos de uma classe.

Usamos o padrão Singleton quando queremos que haja apenas uma instância de uma classe. Para isso, o construtor da classe deve ser privado e a recuperação da instância única deve ser feita por um método público da classe - um método estático.

Na classe `App\Model\Monitor` , o objeto único é criado ou recuperado pelo método estático `getInstance()` . A instância única é armazenada em um atributo `$instance` . Uma vez criado, o objeto fica armazenado na instância até que o programa PHP seja encerrado.

```
    /** @var Monitor **/
    private static $instance;

    private function __construct()
    {
```

```
    }

    public static function getInstance()
    {
        if (self::$instance == null){
            self::$instance = new self();
        }
        return self::$instance;
    }
```

Para que limitar a criação de instâncias? Para que usemos só o que precisamos. Objetos ocupam espaço em memória. Se você cria dois objetos, mas podia fazer o mesmo trabalho com um só, está desperdiçando recursos. Use o necessário, somente o necessário, como diria o urso Balu do filme *Mogli* da Disney (baseado no *Livro da Selva*, de Rudyard Kipling).

> IMPORTANTE! PRINCÍPIO DE BOA ARQUITETURA!
>
> Não importa se os recursos computacionais disponíveis parecem amplos, eles são limitados e têm um preço. A computação é a matemática limitada. Na matemática existe infinito, na computação não. Use somente o necessário e economize recursos.

Os padrões de projeto apresentam soluções para problemas recorrentes de aplicações orientadas a objetos. Alguns padrões já se encontram implementados em componentes de frameworks, mas eventualmente precisamos implementar um padrão para cobrir uma funcionalidade para a qual não há um componente pronto.

Os padrões de projeto, assim como as classes na Orientação a Objeto, podem ser **estendidos**. Quer dizer, você pode ampliar o que já está definido para resolver um problema particular.

Isso quer dizer que, se você tiver a necessidade de limitar uma classe a criar duas instâncias, por exemplo, pode usar o mesmo raciocínio do padrão Singleton. Você não vai encontrar um catálogo com um padrão de projeto para limitar uma classe a duas instâncias, nem três ou quatro. Não é necessário que haja, porque a estratégia é a mesma. Você criará uma variável privada na classe para cada instância e um método público para retornar cada uma.

Limitações de instância não são eventos raros, elas ocorrem devido a regras de negócio. Por exemplo, no xadrez só há dois jogadores. Então a classe *Jogador* em um programa de xadrez só pode ter duas instâncias. Qual o padrão de projeto que resolve esse problema? O Singleton, adaptado ao caso de duas instâncias.

A visão em duas etapas do Mezzio

Bem, falamos sobre o controlador (a classe de sufixo *Handler*), a visão (os arquivos no diretório `templates` e o modelo, especificamente a classe `Monitor`). Agora, precisamos acrescentar algo com relação a forma como a camada da visão é tratada em uma aplicação Mezzio.

O Mezzio usa o padrão **visão em duas etapas** para montar as páginas HTML. Os arquivos referidos pelas classes *Handler* no objeto de resposta representam apenas o corpo da página HTML. A estrutura do documento fica em um arquivo separado em `src/App/templates/layout`, chamado `default.phtml`.

As duas etapas se referem à forma como a página HTML é renderizada. Na primeira etapa, o arquivo indicado pela classe *Handler* é encaixado no interior da estrutura de documento HTML, que está definida no arquivo `default.phtml`. Na segunda etapa, o documento HTML completo é interpretado e devolvido como resposta.

O padrão *visão em duas etapas* facilita a gestão da identidade visual de uma aplicação web. Ao colocar a estrutura do documento HTML

(o cabeçalho da página e os delimitadores do corpo) em um único arquivo, ele centraliza o controle de eventos (JavaScript) e a aparência das páginas (CSS).

Por exemplo, em uma aplicação com 80 páginas HTML, quando for necessário mudar a aparência das páginas, basta alterar um único arquivo, aquele que define o layout.

> IMPORTANTE! PRINCÍPIO DE BOA ARQUITETURA!
>
> Uma boa arquitetura evita mudanças desnecessárias. Crie meios para reduzir o trabalho quando tiver de alterar alguma coisa em um software. Ter mais trabalho para fazer mudanças não é sinal de produtividade, é sinal de arquitetura ruim.

Após a composição da página da rota **home**, temos enfim sua exibição no navegador, conforme a figura a seguir.

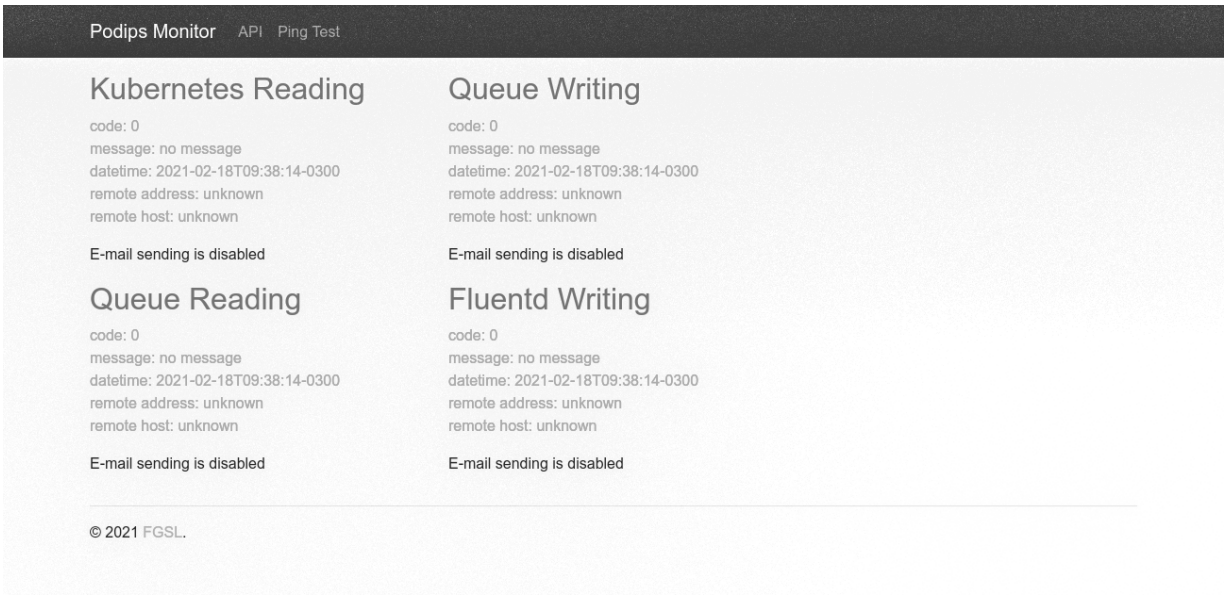

Figura 6.2: A homepage da aplicação podips-monitor.

Existem outros *Handlers* no podips-monitor, mas não há necessidade de analisá-los em profundidade aqui. Basta saber que eles recebem um código de status HTTP dos programas podips-

reader e podips-writer e gravam esse status em arquivos de texto, que serão lidos pelo controlador `App\Handler\HomePageHandler` e exibidos pela combinação dos arquivos `home-page.phtml` (corpo do documento) e `default.phtml` (cabeçalho do documento). Os demais *Handlers* não exibem páginas HTML, com exceção do `App\Handler\ApiPageHandler`, mas apenas dados no formato JSON.

6.4 Documentação da API

O controlador `App\Handler\ApiPageHandler` é responsável apenas por exibir a página de documentação da API, que utiliza a linguagem Swagger. A página no padrão do Swagger exibe os endpoints da aplicação (os URLs que podem ser requisitados a ela) com comentários sobre como devem ser consumidos e ainda contém formulários para testar as requisições. A página da API do podips-monitor é exibida na figura a seguir:

Figura 6.3: Página de documentação da API do podips-monitor.

IMPORTANTE! PRINCÍPIO DE BOA ARQUITETURA!

Se você não vai jogar seu software fora, documente-o. Segundo Roger Pressman, autor do clássico *Engenharia de Software*, a documentação é parte integrante do software. Ter um software sem documentação é como ter um sabre de luz e não ser Jedi: você não sabe muito bem como usá-lo.

No caso de quem desenvolve, a falta de documentação afeta a capacidade de manutenção do software e dificulta a integração com outros softwares. A documentação não precisa ser extensa, mas deve conter o mínimo necessário para usar e modificar o software.

Essa página de documentação é gerada a partir de blocos de documentação, nas classes *Handler*. Esses blocos seguem o padrão de anotações do componente **swagger-php**, cuja documentação está disponível em https://github.com/zircote/swagger-php/blob/master/docs/Getting-started.md.

Uma vez criadas as anotações, basta executar um script na linha do comando para criar a página HTML da API usando o componente **swagger-docs**, disponível em https://github.com/fgsl/swagger-docs.

Por exemplo, a classe `App\Handler\KubernetesHandler` tem uma anotação que documenta o endpoint tratado por aquela classe, descrevendo o URL com as partes fixas e os parâmetros, o método HTTP e o tipo de mensagem. A anotação `@OA\Get` define que o método de envio da requisição é HTTP GET. A anotação `@OA\Parameter` define os dados de um parâmetro do URL, indicado entre chaves. A anotação `@OA\Response` define qual o código HTTP esperado como retorno e descreve o conteúdo devolvido. A seguir vemos a íntegra desse bloco de código:

```
 * @OA\Get(
 *     path="/kubernetes/{code}/{message}",
 *     @OA\Parameter(
 *         name="code",
 *         in="path",
 *         description="código de status HTTP",
 *         required=true,
 *         @OA\Schema(type="string")
 *     ),
 *     @OA\Parameter(
 *         name="message",
 *         in="path",
 *         description="mensagem de status",
 *         required=true,
 *         @OA\Schema(type="string")
 *     ),
 *     @OA\Response(response="200", description="verifica se o podips está
lendo do Kubernetes")
 * )
```

Para gerar a página de API a partir desses blocos de anotações, executamos o seguinte comando no diretório raiz da aplicação:

```
vendor/bin/fsd
```

O comando `fsd`, parte do componente swagger-docs, é instalado pelo Composer com o seguinte comando:

```
composer require fgsl/swagger-docs
```

IMPORTANTE! PRINCÍPIO DE BOA ARQUITETURA!

Automatize a geração da documentação do software. A documentação deve fazer parte do próprio software de modo que a cada liberação de uma versão ela seja criada simultaneamente.

Quando falamos de criar a documentação do software, estamos falando de reunir a documentação de cada parte do software, o que deve ser feito pela pessoa desenvolvedora ao criar ou alterar aquela parte, em uma documentação única, concisa, indexada e organizada, disponível preferencialmente em formato digital em um repositório acessível a todos os interessados.

Não é nossa intenção fazer um curso de programação PHP neste livro. Abordamos o que foi necessário para a análise do programa podips-monitor, cobrindo quase todo o código-fonte, exceto as classes *Handler*, que não eram relevantes para nossa análise.

A linguagem PHP, assim como Go, possui documentação aberta disponível e livros específicos sobre ela para que você possa se aprofundar em suas características particulares.

O código-fonte do programa podips-monitor está disponível para ser livremente estudado, copiado e modificado. Você pode instalá-lo sem a necessidade de que os outros microsserviços estejam ativos se quiser apenas testar a homepage e as saídas dos demais endpoints.

Nosso objetivo aqui foi utilizar o programa **podips-monitor**, que faz a monitoração dos programas **podips-reader** e **podips-writer**, para comentar aspectos arquiteturais de um software de monitoração, além de abordar a questão de reúso com o auxílio de frameworks.

Prosseguiremos nossa análise arquitetural no próximo capítulo, abordando o programa que atua como elemento de controle do

sistema.

Capítulo 7
O microsserviço agendado

E que o mínimo que a gente faça seja, a cada momento, o melhor que afinal se conseguiu fazer. - Lya Luft.

Chegamos ao último microsserviço de nosso sistema de auditoria, o programa executado repetidamente mediante agendamento, o **podips-cronjob**.

Ao contrário do **podips-reader**, do **podips-writer**, do **podips-monitor** e até do **ActiveMQ**, que são softwares livres e abertos, o **podips-cronjob** não tem código-fonte disponível em repositório digital. Mas aqui, neste livro, você terá acesso exclusivo ao código-fonte dele.

Como assim? Você pode perguntar qual é o trabalho de colocar tal programa no GitHub. Bem, na verdade a questão não é o trabalho, mas a relevância de colocar tal código lá. O podips-cronjob é a parte mais flexível do sistema, a mais independente de implementação e também a menor, cabendo integralmente neste capítulo. Isso não é uma maldade para que você tenha de digitar o código-fonte em vez de copiar, mas um modo de enfatizar que você deve **compreender** o que o código-fonte faz.

IMPORTANTE! BOA PRÁTICA DE DESENVOLVIMENTO DE SOFTWARE!

Quando você quiser compreender o que um código-fonte faz, proceda de forma similar à que você usaria ao cruzar uma linha férrea. Quando você está para cruzar uma estrada de ferro, sempre há uma placa com os seguintes dizeres: PARE, OLHE E ESCUTE.

Quando quiser saber o que um código-fonte faz, PARE, LEIA E CONSULTE AS REFERÊNCIAS. Quando dizemos para parar, significa que você deve respirar e planejar sua leitura. Identifique com calma a ordem em que você precisa ler. Faça anotações.

Você pode reduzir a quantidade de partes a serem lidas se estudar primeiro a estrutura do programa em alto nível. Não leia de forma aleatória, pois você perderá mais tempo.

Você pode executar o código-fonte como parte das ações para compreender o que ele faz. Mas se você ler primeiro e consultar as referências das partes que não entendeu da leitura, pode reduzir o número de tentativas de execução necessárias para compreender o funcionamento do programa.

Copiar um código-fonte sem entender como ele funciona é como ter o chapéu do Presto, da Caverna do Dragão. Se você não conhece esse personagem, não se preocupe, vamos fazer uma pausa para um momento cultural. Presto era o mago do grupo de crianças que foi parar em um reino medieval e místico após entrar em uma montanha-russa temática. Presto tinha um chapéu de onde podia retirar objetos de qualquer forma ou substância. O problema é que Presto não sabia como usar o chapéu, e geralmente tirava dele coisas que não queria, o que era muito ruim nos momentos de perigo. No caso do software, usar um código-fonte copiado sem entendê-lo pode ser igualmente ruim na hora de dar suporte.

Bem, você deve ter percebido que a omissão deliberada do código-fonte do **podips-cronjob** foi nossa forma de fazer você prestar atenção à questão de copiar e colar códigos. O fato é que atualmente há uma grande disponibilidade de código-fonte das mais variadas linguagens de programação em diversos repositórios de código aberto e fóruns de discussão.

Não há problema em reaproveitar código-fonte existente. O reúso de software faz parte da busca de produtividade (ao evitar trabalho) e da melhoria da qualidade (ao ter a oportunidade de aperfeiçoar algo existente).

O problema é quando você copia e cola código-fonte sem prestar atenção ao que está fazendo. É como entregar a primeira pizza que estiver pronta no forno ao cliente sem perguntar qual pizza ele quer. Eventualmente você pode entregar a pizza desejada, mas não há garantia disso.

Esperamos que agora você tenha vencido sua ansiedade de copiar códigos-fonte e reflita sobre o programa que é o protagonista deste capítulo.

7.1 O podips-cronjob

Você deve ter compreendido que as partes fundamentais do sistema, aquelas que o fazem *funcionar*, são o trio **podips-reader**, que lê do Kubernetes e escreve na fila; a fila, implementada aqui pelo **ActiveMQ**; e o **podips-writer** que lê da fila e escreve no Fluentd. Esses três compõem o processamento de dados do negócio do sistema.

No capítulo anterior, vimos o podips-monitor, uma parte do sistema referente à monitoração, necessária para garantir a disponibilidade. Neste capítulo, veremos uma outra parte também necessária para o programa de monitoração, o **podips-cronjob**.

O podips-cronjob é um *fingidor*. Ele finge ser um dos programas auditados pelo sistema podips. O mais importante não é a implementação do podips-cronjob, mas o papel que ele desempenha no sistema.

Ele não é um dublê como são os programas **mock-producer** e **mock-consumer**. O dublê substitui um programa para que o funcionamento do sistema como um todo possa ser testado. O fingidor não faz parte dos testes, que podem ser realizados a qualquer momento para simular o sistema e têm obrigatoriamente de ser realizados antes de qualquer nova versão do sistema ser publicada em ambiente de produção.

Definindo o intervalo entre os eventos da aplicação

Para entendermos o papel do podips-cronjob, precisamos tratar de um pequeno detalhe do podips-monitor: o tempo mínimo esperado entre os envios de dados dos programas do núcleo funcional para o programa de monitoração.

O podips-monitor espera que os programas enviem dados em um intervalo de tempo que não pode ser inferior a cinco segundos. Essa regra encontra-se no método `getColor()`, que determina a cor da mensagem, da classe `App\Handler\HomePageHandler`.

Espere aí, se é uma regra, não deveria estar em uma classe da camada de modelo? Sim, se for um atributo do negócio que pode mudar, deveria estar em uma classe dessa categoria. Mas essa regra não deve ser mudada. Ela faz parte de uma condição mínima de funcionamento das aplicações, sem flexibilização.

Assim como é desejável deixar configurável tudo o que puder ser alterado, também é desejável que não seja configurável aquilo que não deve ser alterado. Nós podemos querer que algumas coisas não sejam alteradas. É como determinar que algumas classes não devem ser estendidas, embora tenhamos a possibilidade de estendê-las por herança. Algumas partes de um software podem ser

fixas, porque sua mudança pode comprometer a integridade do sistema.

A questão é que o tempo de intervalo mínimo é um parâmetro de comparação. É como um referencial na física, que indica se você está parado ou em movimento. Você precisa fixar o referencial para poder analisar a sua posição.

Quando você está dentro de um trem, sentado em um assento, você está parado em relação às demais pessoas, mas em movimento em relação a um poste na rua que segue paralelo à linha férrea. Para responder à pergunta "você está parado ou se movendo?", você tem de escolher um dos referenciais. A pergunta será relativa a um deles.

No caso do nosso sistema, nós usamos o tempo de cinco segundos como um referencial. Determinamos que esse intervalo é uma condição de referência para saber se há algum problema no sistema ou não. Esse intervalo funciona para nós como se fosse uma lei da natureza, mas apenas no universo do nosso sistema. Em nosso microuniverso, o Kubernetes não passa menos de cinco segundos sem gerar eventos de mudança dos IPs dos pods.

Para você entender mesmo a questão dos cinco segundos serem fixos no sistema, antes de se revoltar querendo movê-los para uma classe modelo, pense na questão da febre. Você tem um termômetro e quer saber se alguém está com febre. Para isso, você precisa determinar uma temperatura que representa o limiar entre o estado normal e o estado febril. Usamos a temperatura de 38°C como referencial. Essa temperatura não muda, pois faz parte da natureza do nosso sistema. Não é algo que será modificado por um decreto do governo ou pela mudança de gostos de consumidores. É um fato bem estabelecido.

As regras de negócio que se encontram na camada de modelo do MVC são regras dadas pelo cliente. Essas regras são mutáveis. São regras do microuniverso dele. Sabendo que podem mudar, devemos

facilitar a mudança colocando essas regras em um local onde podemos modificá-las com facilidade.

Por outro lado, as regras imutáveis não devem ser fáceis de mudar. O sistema deve ter mecanismos para enfatizar que elas não são parâmetros configuráveis, mas fatos incontestáveis. São como os axiomas na geometria. Você não os discute, apenas aceita.

Mas um intervalo maior que cinco segundos entre os eventos é sempre um problema? A resposta é não. Pode ser que em algum momento passem sete segundos entre o envio de eventos pelo Kubernetes e que não haja nada de errado. Ele pode demorar três segundos em um momento e dez segundos em outro, mas geralmente leva menos de cinco.

Podemos ter anomalias, que não serão significativas a não ser que ocorram em grande quantidade, mas isso só pode ser concluído a partir da análise de um período. Não podemos achar que o sistema está falhando porque em apenas uma determinada hora do dia ele levou mais de cinco segundos entre uma leitura de eventos e a seguinte.

Evitando alarmes falsos

O que fazemos então para evitar que as exceções não significativas produzam falsos alertas? Nós usamos um **placebo**!

Um placebo, na medicina, é um elemento usado como base de comparação para testar a eficácia de medicamentos. Para sermos mais precisos, ele serve para eliminar falsas indicações de eficácia.

Pense, por exemplo, em uma pessoa que diz que comer Chokito cura dor de garganta. Ela afirma que estava com dor de garganta, comeu um Chokito e a dor passou no dia seguinte. Para ela, isso indica que o Chokito a curou. Mas a ciência não funciona assim.

Quando todos estivessem curados, faríamos a análise dos resultados e, ao comparar a quantidade de pessoas curadas

imediatamente com as não curadas e contar quantas comeram Chokito e quantas não comeram, poderíamos determinar estatisticamente o quão determinante o Chokito foi na cura. Se as quantidades fossem distribuídas de forma que não houvesse concentração de curados entre os que comeram Chokito, poderíamos concluir que ele é um elemento aleatório na cura. Não precisamos que todos que comeram Chokito sejam curados para prosseguir no estudo dele como remédio, mas precisamos que haja uma maioria significativa, para indicar uma relação e excluir o acaso.

O tablete falso é o parâmetro de comparação para determinar a nulidade da hipótese do Chokito ser um medicamento eficaz contra a dor de garganta. No caso do sistema podips, que a esta altura você já deve ter esquecido que existe, o programa podips-cronjob serve como parâmetro de comparação para determinar se há uma falha no Kubernetes.

Assim como queremos evitar um alarme falso de que o Chokito cura dor de garganta para que as pessoas não corram inutilmente para as bombonières, queremos evitar um alarme falso do podips-monitor que provoque uma correria da equipe de suporte do podips em busca de uma causa de falha.

Esse é o papel do podips-cronjob. Ele é nosso tablete falso, nosso placebo. É nosso parâmetro de comparação. E como ele atua? Da seguinte forma: a cada 60 segundos, ele é executado no ambiente do Kubernetes.

Ele roda rapidamente, fazendo uma simples requisição HTTP, mas sua execução obriga o Kubernetes a atribuir-lhe um IP e, consequentemente, a reportar esse evento, que então será lido pelo **podips-reader**. Ele garante que pelo menos a cada 60 segundos há uma alteração de IP no Kubernetes e que a cada 60 segundos haverá pelo menos um evento reportado.

Assim nós excluímos a falha do Kubernetes das possibilidades de falha caso os outros programas hospedados não reportem mudança dentro do intervalo de um minuto.

Mas o intervalo mínimo não é cinco segundos? Sim, mas não sabemos quando a interrupção de eventos dos outros programas (os programas reais) vai acontecer. Ela pode ocorrer um segundo após a execução do podips-cronjob, cinco segundos depois, dez segundos depois etc. é aleatório. Assim como a pessoa que comeu o tablete falso pode ficar curada um dia depois, dois dias depois ou uma semana depois.

O podips-cronjob não garante que não há nenhum problema com o sistema podips. Ele apenas exclui uma possibilidade de falha para que possamos reduzir o espaço de busca das causas de problemas quando eles ocorrerem, ou a própria busca de problemas. Queremos eliminar os alarmes falsos.

É como se, no mundo do software, estivéssemos evitando trotes. Um alarme falso pode levar uma equipe de suporte ao desespero sem motivo algum. E se os alarmes falsos não forem reduzidos, chegará uma hora em que a equipe não dará atenção aos alarmes verdadeiros, como na história do pequeno galinho que sempre saía gritando pela cidade, dizendo que o céu estava caindo.

> IMPORTANTE! PRINCÍPIO DE BOA ARQUITETURA!
>
> Evite alarmes falsos em um sistema de software. Alarmes falsos só nos fazem perder tempo.

Depois de falar de magia, febre e chocolate, sem deixar de explicar entretanto o papel do podips-cronjob no sistema podips, vamos comentar a implementação desse programa na seção seguinte.

7.2 Implementação do podips-cronjob

O podips-cronjob é um programa escrito em JavaScript. Para o que ele faz, não há realmente nada específico que exija a linguagem JavaScript. A escolha da linguagem se deve a uma restrição de ambiente.

A equipe que administra o agendamento de aplicações no cluster Kubernetes, onde o podips é executado, só permite o agendamento de aplicações feitas em Java e JavaScript. Por que eles fizeram isso? Porque até o momento da implantação do sistema de administração do cluster Kubernetes, só havia aplicações agendadas nessas linguagens. Então eles assumiram que só haveria essas duas e fixaram as opções.

Isso é uma falha de arquitetura, se pensarmos que em um contexto de microsserviços deve haver diversidade de linguagens, mas ela é tolerada porque a demanda por aplicações agendadas é muito baixa.

O podips-cronjob consiste em um arquivo JavaScript chamado `index.js`, que é executado por um servidor NodeJS. NodeJS é um interpretador de JavaScript, que permite usar essa linguagem, projetada originalmente para ser executada pelos navegadores web, para construir o back-end de aplicações web.

NodeJS permite que JavaScript seja usado nos dois lados de uma arquitetura cliente-servidor, o lado que envia a requisição e o lado que envia a resposta. NodeJS é um software livre e aberto, disponível em https://nodejs.org.

O arquivo `index.js` começa com a definição de uma constante `HOST`, que lê a variável de ambiente `PODIPS_HOST` para saber o endereço do podips-monitor.

```
const HOST = process.env.PODIPS_HOST
```

Em seguida, importamos um módulo do NodeJS que encapsula objetos que permitem a comunicação com o protocolo HTTP seguro.

```
var https = require('https');
```

Depois, criamos uma variável `host` para armazenar o conteúdo da constante `HOST`. Você deve estar se perguntando qual é o motivo disso. É que, ao criar uma variável do tipo texto no JavaScript, ela se torna automaticamente um objeto, que tem métodos para manipular texto, e nós precisaremos fazer uma manipulação, conforme veremos mais à frente.

```
var host = HOST;
```

O próximo passo é configurar um array com as opções da requisição HTTP que o programa fará. Nessa configuração, passamos o endereço do podips-monitor, sem o protocolo, especificamos a porta de comunicação, indicamos que a requisição não verificará o certificado (pois o podips-cronjob não tem certificado), definimos o endereço que será requisitado e o método HTTP.

```
var options = {
  host: host.replace('https://',''),
  port: '443',
  rejectUnauthorized: false,
  url: HOST + '/podips/all',
  method: 'GET'
};
```

Uma vez definidas as opções da requisição HTTP, usamos o método `request()` do objeto `https` para fazer uma requisição HTTP GET para o podips-monitor, de um endpoint que retorna todos os status do sistema podips em formato JSON. Em caso de sucesso, exibimos a resposta no console do podips-cronjob, de modo que um administrador de sistema pode acompanhar os resultados das requisições.

Observe, no trecho de código a seguir, que o controle das respostas à requisição é feito por programação orientada a eventos. Existem dois eventos tratados pelo objeto de resposta `res`, que é o segundo argumento do método `request()`:

- `data`: esse evento é disparado quando a requisição retorna um código HTTP 200, indicando que houve sucesso e há um resultado válido.
- `error`: esse evento é disparado quando a requisição retorna um código de erro HTTP.

Os eventos são tratados pelo método `on()` do objeto de resposta. Esse método recebe o nome do evento como primeiro argumento e o objeto de erro como segundo argumento. Em caso de erro na resposta, nós imprimimos o objeto de erro.

Todo o bloco de envio da requisição está dentro de uma estrutura de tratamento de exceção (`try...catch`). Caso haja falha no envio, também será impressa uma mensagem. São dois casos distintos de falha. Em um caso, o programa pode fazer a requisição HTTP e receber uma resposta com status de erro. No outro, ele pode falhar em fazer a requisição.

A seguir temos o bloco de código do envio da requisição:

```
try {
  var req = https.request(options, res => {
      console.log('Run HTTPS request');
      res.on('data', d => {
        console.log('Run HTTPS response');
        process.stdout.write("Podips Monitor output:\n");
        process.stdout.write(d);
      })
    })

    req.on('error', error => {
      console.error(error);
    })
```

```
    req.end();
}
catch (e) {
    console.log(e);
}
```

SINTAXE DO JAVASCRIPT: ALGUNS DETALHES

JavaScript é uma linguagem de programação cuja sintaxe é similar em vários aspectos à linguagem C e em outros à linguagem C++. É uma implementação da especificação ECMA-262, cuja documentação está disponível em https://www.ecma-international.org/publications-and-standards/standards/ecma-262.

Apesar do nome, JavaScript não tem qualquer ligação ou dependência com a linguagem Java, a não ser o oportunismo da Netscape em tentar vincular sua criação com o produto da Sun Microsystems. É isso mesmo, o nome foi uma jogada de marketing.

Blocos de código em JavaScript são delimitados por chaves ({}).

As instruções em JavaScript terminam em ponto e vírgula (;). Você pode colocar várias instruções em uma mesma linha lógica de arquivo, o que faz com que o interpretador saiba que a instrução terminou é o ponto e vírgula.

Funções em JavaScript são blocos de código com o nome `function`. Classes em JavaScript também usavam a palavra `function`. Em 2015, a implementação ES6 de JavaScript passou a permitir a palavra `class` para definir classes.

JavaScript é uma linguagem de programação multiparadigma. Você pode, em um mesmo arquivo JavaScript, usar classes (paradigma orientado a objetos), funções (paradigma estruturado) e código monolítico (sem paradigma) fora de classes e funções.

JavaScript pode reaproveitar código de classes pelo mecanismo da herança.

JavaScript foi projetado originalmente para ser usado nos navegadores, para gerar conteúdo dinâmico e controlar eventos

no lado do cliente. Atualmente, é possível usar JavaScript no lado servidor e na administração de bancos de dados MongoDB, além de escrever script para automação de infraestrutura.

O tipo das variáveis em JavaScript é definido dinamicamente, mas diferente de PHP, elas são declaradas com o operador `var` ou `let`. Variáveis criadas com `var` têm escopo global, enquanto variáveis criadas com `let` têm escopo de bloco de código.

Após serem criadas, as variáveis em JavaScript são fracamente tipadas, o que significa que é possível fazer operações diferentes do tipo da variável. Mas ao contrário de PHP, JavaScript não tem comportamento uniforme de acordo com a operação. Em PHP, se você tentar somar "1" com 1, o resultado é 2, e se tentar subtrair 1 de "1" de 1, o resultado é 0. Em JavaScript, se você tentar somar "1" com 1, o resultado é "11" e se tentar subtrair 1 de "1", o resultado é 0! Ou seja, em JavaScript você precisa cuidar para que os operandos sejam do tipo que você espera antes de fazer operações.

O motivo do podips-cronjob ter esse nome é porque ele é executado pelo seu cadastro em uma tabela de cronjobs, literalmente *trabalhos cronometrados*. Essa tabela é uma agenda do sistema operacional, que indica programas que têm de ser executados em intervalos regulares.

Você pode dizer que um programa tem de ser executado a cada mês, semana, dia, hora, minuto ou até segundo. No sistema operacional Linux, a tabela de cronjobs fica no arquivo `/etc/crontab`.

```
# Exemplo de definição de um cronjob:
# .---------------- minuto (0 - 59)
# |  .------------- hora (0 - 23)
# |  |  .---------- dia do mês (1 - 31)
# |  |  |  .------- mês (1 - 12) OU jan,feb,mar,apr ...
# |  |  |  |  .---- dia da semana (0 - 6) (Sunday=0 ou 7) OU
sun,mon,tue,wed,thu,fri,sat
```

```
#|  |  |  |  |
# *  *  *  *  * [no do usuário] [comando a ser executado]
```

Uma vez cadastrado na tabela de cronjob, o sistema operacional se encarregará de executar o programa no horário e dia programados. É assim que o podips-cronjob é executado a cada 60 segundos.

Capítulo 8
Conclusão

O que vale na vida não é o ponto de partida e sim a caminhada. Caminhando e semeando, no fim terás o que colher. - Cora Coralina.

Encerramos assim nossa caminhada por princípios de (boa) arquitetura de software, contextualizados para o universo dos microsserviços.

Uma boa arquitetura de software conduz a construção de softwares que sobrevivem a constantes alterações, que incluem adição de novas funcionalidades, correções de erro, evoluções provocadas por necessidade ou oportunidade e integrações com outros sistemas.

Uma boa arquitetura é vital para a **sobrevivência do software** se ele tiver um ciclo de vida de longo prazo. Se você vai construir software para durar uma semana, talvez possa abdicar de arquitetura, mas, ao fazer isso, pode perder a oportunidade de reaproveitar componentes para outros projetos.

Uma boa arquitetura de software faz com que seja possível continuar dando manutenção a um software mesmo que a cada mudança ele fique mais complexo. Uma boa arquitetura equilibra a complexidade com a facilidade de compreensão.

Talvez você pergunte: é só isso? Quer dizer, com o que vimos aqui conseguiremos definir uma boa arquitetura para todos os projetos de software daqui em diante? A resposta é não.

Você viu que abordamos alguns padrões de projeto em nossa caminhada. É interessante que você procure ler a respeito de mais padrões de projeto, pois eles contêm a experiência de desenvolvedores para problemas recorrentes. A literatura sobre padrões de projeto é abrangente e você encontra livros com

implementações deles em diversas linguagens. Na Casa do Código, por exemplo, você pode encontrar os seguintes livros sobre padrões, aplicados respectivamente para Ruby, Java e PHP:

- Refatorando com padrões de projeto: Um guia em Ruby.
- Refatorando com padrões de projeto: Um guia em Java.
- Design Patterns com PHP 7: Desenvolva com as melhores soluções.

O site https://sourcemaking.com traz, em inglês, a descrição de cada um dos padrões de projeto mencionados no livro clássico *Padrões de Projeto* da Gangue dos Quatro (Erich Gamma e seus incríveis amigos). Este site também inclui os antipadrões de projeto, que são lições sobre o que não fazer ou evitar na construção de código-fonte Orientado a Objetos.

No site https://oasisbr.ibict.br, você pode encontrar trabalhos de conclusão de curso, dissertações e teses, todas em português, sobre padrões de projeto implementados em várias linguagens. Basta selecionar as palavras-chave e fazer a busca.

Há um conjunto de princípios chamado **Calistenia de Objetos**, que traz orientações para melhorar programas escritos com o paradigma da Orientação a Objetos. Você pode ler mais a respeito desses princípios no seguinte artigo:

https://devsquad.com.br/dicas-importantes-que-farao-aumentar-a-qualidade-dos-seus-codigos/

Para terminar, podemos citar também os *Dez Mandamentos de um Desenvolvimento de Software com Sucesso*, de Marc Hamilton:

1. *Comece o desenvolvimento com requisitos de software*. Construa código somente a partir de requisitos. Idealmente, construa usando desenvolvimento orientado a testes, que garante que o requisito fará parte de um código-fonte. Mais idealmente, construa usando desenvolvimento orientado a comportamento, para os casos em que testes com requisitos

são definidos junto com o cliente, no levantamento de requisitos.

2. *Honre seus usuários e comunique-se com eles frequentemente*. A maior parte dos problemas na vida ocorre por falta de comunicação. Se tiver dúvidas, pergunte. Antes de prosseguir, mostre o que fez até o momento e confirme se está certo. Os princípios do desenvolvimento ágil valorizam bastante a comunicação, porque ela evita que se trabalhe em algo diferente do que é esperado pelo cliente.

3. *Não permita mudanças de requisitos sem garantias*. A mudança é bem-vinda, como afirmam os princípios do desenvolvimento ágil, mas não é por isso que você vai aceitar qualquer coisa do seu cliente. As mudanças precisam ser analisadas e negociadas, porque têm impacto no produto final.

4. *Invista antecipadamente em arquitetura de softwar*e. Não espere o projeto começar para pensar em arquitetura. Procure componentes reutilizáveis e estude-os, preparando-se para demandas futuras. Faça laboratórios de simulação de projetos. Exercite-se, não somente sozinho, mas em equipe.

5. *Não confunda produtos com padrões*. Há muitos componentes de software disponíveis para os quais você pode terceirizar trabalho. Mas não programe para implementações específicas, programe para interfaces. Defina padrões de comunicação para que seja possível substituir componentes quando for necessário sem grande impacto para o software.

6. *Reconheça e retenha seus maiores talentos*. Se você está na posição gestora, precisa fazer a gestão de conhecimento de sua equipe. Como arquiteto de software, você deve aproveitar o que cada um tem de melhor e também motivar cada um a melhorar o que sabe. Capacite no que sua empresa precisa, mas também procure promover capacitação no que interessa

aos membros da equipe e tente aproveitar conhecimento não utilizado em inovação.

7. *Compreenda tecnologia orientada a objetos*. O tempo passa, o tempo voa e o paradigma da Orientação a Objetos continua numa boa. Na verdade, a maior parte dos serviços que temos hoje está implementada em modelos de objetos. Saber programação Orientada a Objetos é obrigatório não somente para desenvolvedores e desenvolvedoras hoje em dia, mas também para pessoas que administram sistemas, pois diversos softwares de infraestrutura trabalham com arquivos de configuração baseados em objetos.

8. *Projete aplicações centradas em web e componentes reusáveis*. Estamos no mundo das aplicações distribuídas. Mesmo que sua aplicação seja inicialmente construída para ser desktop, instalada e operada em somente uma máquina, fazendo uso direto dos recursos do sistema operacional, pode ser que ela se beneficie no futuro de uma integração com serviços web. Então é bom pensar nisso ao definir sua arquitetura. A arquitetura também deve buscar terceirizar o máximo de trabalho para componentes reusáveis e gerar o máximo de componentes reusáveis para projetos futuros.

9. *Planeje para mudança*. Lembra da **configurabilidade** e da **parametrização**? Evite criar definições fixas no software. Deixe-o sempre aberto para alterações que possam ser feitas sem a necessidade de lançamento de uma nova versão. O software deve ser desenvolvido de forma que possamos alterar seu comportamento o máximo possível dentro de uma mesma versão.

10. *Implemente e adira a um processo de aceitação de produção*. A qualidade deve ser **verificável**. Para isso, é necessário usar métricas que indiquem se a estrutura do código-fonte tem uma qualidade aceitável. Isso pode ser feito com as ferramentas de teste, que geram relatórios de cobertura de teste, e ferramentas

de análise de complexidade, que mostram se há problemas estruturais no software.

Finalmente, não se esqueça: continue estudando sempre!

Capítulo 9
Referências

BASS, Len. CLEMENTS, Paul. KAZMAN, Rick. *Software Architecture in Practice*. 3. ed. Westford: Pearson Education, 2013.

BROOKS, Frederick P. *O mítico homem-mês:* ensaios sobre engenharia de software. Tradução: Cesar Brod. Rio de Janeiro: Elsevier, 2009.

BROWN, Simon. *Software Architecture for Developers*. Leanpub, 2014.

COLIN, Sílvio. *Uma Introdução à Arquitetura*. 5. ed. Rio de Janeiro: Uapê, 2000.

DEVSQUAD. *Dicas importantes que farão aumentar a qualidade dos seus códigos*. 2021. Disponível em: https://devsquad.com.br/dicas-importantes-que-farao-aumentar-a-qualidade-dos-seus-codigos. Acessado em 5 de julho de 2021.

DJIKSTRA, E. W. *The threats to computing science*. ACM South Central Regional Conference. Nov. 1984.

DJIKSTRA, E. W. *A discipline of programming*. Englewood Cliffs: Prentice-Hall, 1976.

FOWLER, M. *Inversion of Control Containers and the Dependency Injection pattern*. 23 de janeiro de 2004. Disponível em: https://www.martinfowler.com/articles/injection.html. Acessado em 13 de abril de 2021.

GAMMA, E. *Erich Gamma on Flexibility and Reuse: A Conversation with Erich Gamma, Part II*. Artima, Mai. 2005. Disponível em: https://www.artima.com/articles/erich-gamma-on-flexibility-and-reuse. Acessado em 13 de abril de 2021.

GAMMA, E. HELM, R. JOHNSON, R. VLISSIDES, J. *Design Patterns*: Elements of reusable object-oriented software. Addison-Wesley, 1995.

HAMILTON, Marc. *Ten Commandments of Successful Software Development*. 12 de outubro de 2001. Disponível em: https://www.informit.com/articles/article.aspx?p=23608. Acessado em 5 de julho de 2021.

HOWARD, M. e LEBLANC, D. *Escrevendo código seguro*: estratégias e técnicas práticas para codificação segura de aplicativos em um mundo em rede. 2.ed. Porto Alegre: Bookman, 2005.

MARANHO, Ricardo. *SGBD de alta escalabilidade com suporte a dados georreferenciados*. Dissertação (Mestrado em Engenharia Informática e de Computadores). Lisboa, Instituto de Engenharia de Lisboa, 2014.

MCCONNELL, Steve. *Code Complete: um guia prático para a construção de software*. 2.ed. Porto Alegre: Bookman, 2005.

PRESSMAN, Roger S. *Software Engineering: A Practioner's Approach*. 7. ed. New York: McGraw-Hill, 2010.

TANENBAUM, Andrew. STEEN, Maarten Van. Sistemas distribuídos. 2. ed. Tradução: Arlete Simille Marques. São Paulo: Pearson Education do Brasil, 2007.

VITRUVIO. *DE ARCHITECTURA*. Londres: Ludgate Hill, 1874.